RL-TR-95-39
Final Technical Report
March 1995

DIGITAL SIGNAL PROCESSING RAPID PROTOTYPING FOR MILITARY COMMUNICATION SYSTEMS

University of Kansas

Srinivas Sivaprakasam, Christos Neophytou, Glenn Prescott,
and Tim Johnson

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

Rome Laboratory
Air Force Materiel Command
Griffiss Air Force Base, New York

This report has been reviewed by the Rome Laboratory Public Affairs Office (PA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

RL TR-95-39 has been reviewed and is approved for publication.

APPROVED: Stephen C. Tyler

STEPHEN C. TYLER
Project Engineer

FOR THE COMMANDER: Henry J. Bush

HENRY J. BUSH
Deputy for Advanced Programs
Command, Control & Communications Directorate

If your address has changed or if you wish to be removed from the Rome Laboratory mailing list, or if the addressee is no longer employed by your organization, please notify RL (C3BB) Griffiss AFB NY 13441. This will assist us in maintaining a current mailing list.

Do not return copies of this report unless contractual obligations or notices on a specific document require that it be returned.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave Blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	March 1995	Final ----

4. TITLE AND SUBTITLE
DIGITAL SIGNAL PROCESSING RAPID PROTOTYPING FOR MILITARY COMMUNICATION SYSTEMS

5. FUNDING NUMBERS
C - F30602-93-C-0109
PE - 62702F
PR - 4519
TA - 42
WU - PE

6. AUTHOR(S)
Srinivas Sivaprakasam, Christos Neophytou, Glenn Prescott, and Tim Johnson

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
University of Kansas
CECASE Laboratory
2291 Irving Hill Drive
Lawrence KS 66045

8. PERFORMING ORGANIZATION REPORT NUMBER
N/A

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Rome Laboratory (C3BB)
525 Brooks Rd
Griffiss AFB NY 13441-4505

10. SPONSORING/MONITORING AGENCY REPORT NUMBER
RL-TR-95-39

11. SUPPLEMENTARY NOTES
Rome Laboratory Project Engineer: Stephen C. Tyler/C3BB/(315) 330-3618

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)
This research effort dealt with developing a digital signal processing (DSP) radio rapid prototyping testbed. The idea was to integrate advanced computer-aided design (CAD) communications software with a multiple DSP processor. A simplified spread spectrum transceiver link was developed and evaluated using the system.

14. SUBJECT TERMS
Spread spectrum, Rapid prototyping, Digital signal processing

15. NUMBER OF PAGES
92

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. Z39-18
298-102

Contents

List of Figures iii

Introduction 1
- 1.1 Background . . . 2
- 1.2 Research Objectives and Approach . . . 3

System set-up 5
- 2.1 Hardware set-up . . . 7
  - 2.1.1 PC-A/D Interface . . . 7
- 2.2 Transmitter Model . . . 9
- 2.3 Receiver Model . . . 9

Optimization 15
- 3.1 Need for optimization . . . 16
  - 3.1.1 Algorithmic optimization . . . 16
  - 3.1.2 Code Optimization . . . 18

Analysis and Results 19
- 4.1 Bit Rate Determination . . . 20
- 4.2 MSK Carrier and Timing Recovery . . . 22
  - 4.2.1 Delay Block . . . 23
  - 4.2.2 Derivative Block . . . 24
- 4.3 Choosing the Bandpass filter . . . 24
- 4.4 Timing Recovery and Choice of Parameters . . . 25
- 4.5 Results . . . 27

Multiprox 29
- 5.1 Standard Multiprox . . . 31
- 5.2 IPC in Standard Multiprox . . . 32
- 5.3 Our IPC Methods . . . 33
  - 5.3.1 The TMS320C40 Processor . . . 34
  - 5.3.2 Parallel Processing with C40 . . . 35
- 5.4 Comparison of the Methods . . . 39

Transmitter 44
hold_true . . . 45
rs232_ccsk . . . 46
framer . . . 48
encoder2 . . . 49
sync_gen . . . 50
data_iq . . . 51
msk_mod . . . 52
carrier . . . 53

Receiver 54
adequalize . . . 55
carrier_rec . . . 57
msk_demod . . . 59
frame_sync2 . . . 61
decoder2 . . . 63
level2bits . . . 65
hold_true . . . 66
rise_edge_true . . . 67

Interprocessor Communication Blocks 68
ipc1_in . . . 69
ipc1_out . . . 71
ipc2_in . . . 73
ipc2_out . . . 75
nullsib . . . 77
nullsob . . . 78
atod_filt . . . 79

List of Figures

2.1 Hardware set-up for Rapid Prototyping 6

2.2 Implementation methodology for Rapid Prototyping 11

2.3 RS-232 serial data timing waveform 12

2.4 RS-232 cable connections used 12

2.5 Transmitter Model 13

2.6 Receiver model 14

4.1 Delay Block Implementation 28

4.2 Derivative Block Implementation 28

5.1 Multiprox and its interactions with the rest of SPW 32

5.2 Quad C40 Board Layout 34

5.3 CPU-DMA Interactions for IPC 36

5.4 Linked List & Operation of DMA 37

5.5 Use of buffers for IPC-1, IPC-2 and S-MPX 38

5.6 Delta Modulation example 42

5.7 SYSTEM-1 with one data transfer used in Table 5.1 43

5.8 SYSTEM-2 with two data transfers used in Table 5.1 43

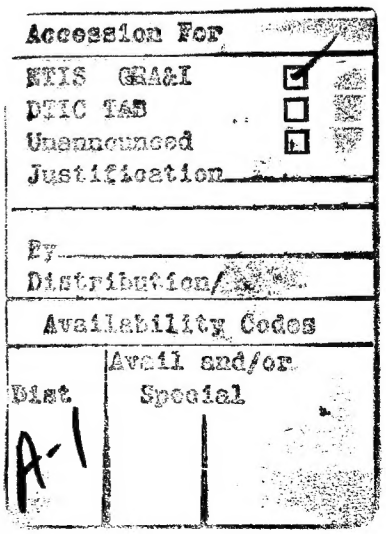
List of Tables

4.1 Timing Details for the Transmitter 27

4.2 Timing Details for the Receiver 27

4.3 BER performance of ideal and simulated systems 27

Introduction



1.1 Background

Digital signal processing techniques offer several significant advantages in communication system design. Precision, long term stability and reduced maintenance costs are characteristics of this technology. Recent introduction of low-cost, highly linear analog-to-digital converters, and DSP microprocessors capable of parallel processing, make digital processing the architecture of choice. Military planners in the Air Force's Speakeasy Radio Development Program have recognized these advantages and have committed a significant portion of the radio design to a DSP-oriented solution.

In order to effectively implement the Speakeasy radio, as well as the next generation military communication systems, researchers and engineers will need development tools to design and evaluate their systems. These tools need to be applied early in the design cycle in order to provide a positive influence on system performance, effectiveness and cost. Essential tools include the following:

a. Simulation-based analysis - Simulation is an important tool in the development of the overall communication system. Analyses can be executed to evaluate the performance of the overall system in a particular environment. For example, waveform vulnerability can be evaluated to determine the relative jam resistance, or covertness of the system to particular threats. The engineer also might want to evaluate the bit error performance of the system, or the behavior of the system over particular channel conditions. Simulation allows the engineers to make a quick *what if* analysis, by changing the error correcting code, or altering the modulation format. It is a tremendously powerful tool for the early stages of the system development cycle. Once the system parameters have been established, the next step is to implement a prototype of the system in order to fine-tune the algorithms and evaluate the performance of the system in real time. In order to complete this step, a DSP development platform is needed in which engineers can evaluate design trade-off's.

b. DSP Prototyping - Digital Signal Processing is becoming the technology of choice for radio systems, primarily because of the flexibility it provides and also because of the precision with which critical communications signal processing functions can be executed. One of the drawbacks of this technology is that implementation of DSP algorithms on special-purpose DSP microprocessors requires the communications engineer to be proficient in the assembly language of the processor being used. While this is an essential requirement for the engineer who develops the production model, it would be beneficial for the systems engineers at an early point in the design cycle

to have the capability to evaluate prototypes which are easily configured. An effective DSP prototyping system would allow the engineers to operate the DSP processor from a higher level, using the essential elements of simulation. The advantage is that the engineer can quickly configure DSP algorithms into complete subsystems which can produce or process signals in real-time. The signal processor can now interface with test equipment and realistic evaluations of algorithm effectiveness, processor speed and processor loading can be observed.

With simulation and prototyping tools in place developers can proceed to focus on the real time implementation and testing of transmitter and receiver signal processing subsystems. The simulation tools for digital radio communications were developed for Rome Labs by the University of Kansas and others under a previous effort. The research reported here was targeted at integrating the software simulation and DSP hardware to produce a system which allows implementation of DSP algorithms in real-time.

1.2 Research Objectives and Approach

For this project we engaged in a 12-month effort during which our research objectives consisted of the following:

a. Study system level approaches for implementing a DSP-based digital radio transmitter and receiver. This effort began with an investigation of effective methodologies for designing DSP-based communications systems - specifically, digital radio transmitters and receivers. We conducted a literature search of currently known techniques, practices and available specialized DSP hardware support. Additionally addressed such important issues as when to trade off the implementation of DSP algorithms from DSP microprocessors to dedicated high speed DSP hardware.

b. Develop a DSP Evaluation and Prototyping System (DSP-EPS) by integrating SPW design tools with a specific DSP signal processor system. This effort focused exclusively on the use of the Texas Instruments TMS320C40 processor. In consultation with Rome Labs, we obtained a specific commercially available TMS320C40 card containing multiple C40 processors. With this card, we developed a fully integrated rapid prototyping system using the advanced DSP system development tools which are an integral part of the Signal Processing Worksystem. The unique capabilities of SPW make it an ideal tool for this unique task. We used the code generating capability of SPW to produce DSP processor-specific code to demonstrate the feasibility of employing SPW as an effective rapid prototyping tool. The Code Gener-

ation System, produces optimized, device specific C-code directly from the SPW Block Diagram Editor; and Multiprox, which is the SPW multiprocessor tool which assigns tasks or subtasks within an algorithm to multiple processors. Development of a rapid prototyping capability using multiprocessing DSP processors also required investigation of DSP processor loading for critical radio algorithms, and assignment of these algorithms to multiple processors using the SPW Multiprox system.

c. Demonstrate an effective multiprocessing rapid prototyping capability using the DSP-EPS. We concluded the research effort with a demonstration of the rapid prototyping system. We exercised its capability by implementing a prototype scaled version of a spread spectrum radio transmitter and receiver using a modulation waveform provided by Rome Labs engineers. This system operated at some nominal intermediate frequency (IF) which is was appropriately scaled. This demonstration established the practicality of real-time DSP implementation of critical signal processing algorithms for the Speakeasy Radio Program. Since SPW is currently in use at Rome Laboratories as a communications simulation tool; with only a modest investment in signal processing hardware, the capability will now exist to develop and test waveforms and algorithms for the Air Force's next generation communication systems.

All research was conducted at the University of Kansas Center for Research using the facilities of the Telecommunications and Information Sciences Laboratory (TISL), and the Center of Excellence for Computer Aided Systems Engineering (CECASE).

System set-up

The goal of this project was to demonstrate the concept of rapid prototyping with an example implementation of the JTIDS modem on DSP's using the SPW environment, to maximize the information bit rate and at the same time maintain high system performance. The JTIDS modem uses 5-32 CCSK spreading followed by MSK modulation at the transmitter and MSK demodulation, frame synchronization and CCSK decoding at the receiver. Some of the key issues involved in the implementation were to come up with efficient algorithms for all the modules of the transmitter and receiver (including efficient carrier, symbol and frame synchronization algorithms at the receiver) and efficient ways of implementing these algorithms in SPW.

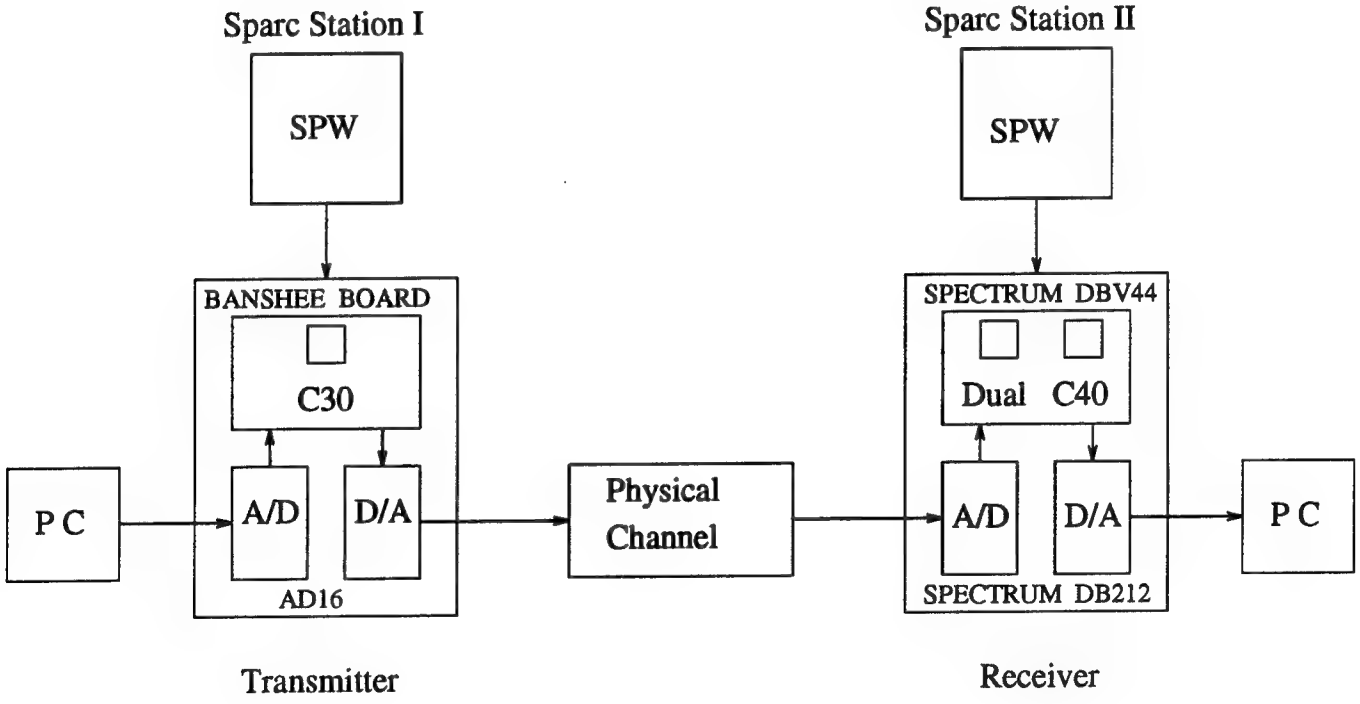


Figure 2.1: Hardware set-up for Rapid Prototyping

2.1 Hardware set-up

Figure 2.1 shows the hardware set-up used for the proposed implementation. The information bit source for system was ascii bits representing text typed on a PC keyboard that are output to the PC serial port by Kermit software. The signal representing this bit stream is sampled by an A/D converter on the ASPI AD-16 Interface Board. The sampled bit stream is then encoded, modulated and up-converted to IF on a C30 DSP. The analog IF signal at the output of the D/A converter is transmitted through a physical channel which gets corrupted by white noise. The analog signal at the receiver is sampled by an A/D converter on a Spectrum DB212 board. The sampled signal is then down-converted, demodulated and decoded to get information bits, which are sent to the serial port of another PC for display as text on the screen.

Figure 2.2 depicts the essence of the rapid prototyping methodology using SPW. Once a system has been defined and critical system parameters identified, performance evaluation of the algorithms may be done using SPW. Once the algorithms have been fine tuned, real-time performance may be assessed using the Code Generation System CGS, which generates real-time code for the TMS320 C30 and C40 DSP's. If satisfactory performance has not been achieved in real-time, the algorithm may be very easily modified or the parameters may be changed at the block level and the whole process repeated.

2.1.1 PC-A/D Interface

Kermit

Characters in a PC are represented as a 7-bit ASCII code. For serial transmission an 8th bit, representing a hardware parity bit (usually 0), is sent as the most significant bit. These bits representing each character typed on the screen are sent to the serial port, LSB first, using Kermit software. To operate Kermit

- Type *kermit* in the directory containing Kermit executables
- Set parameters such as speed of transmission, port number and auto wrap.
- Type connect, press enter and start typing

The above set-up may be used at the transmitter and the receiver. Kermit at the transmitter PC sends characters typed on the screen to the serial port as bits while Kermit at the receiver PC takes incoming bits at the serial port and displays them as characters on the screen.

The RS-232 interface

RS-232 is a standard for serial bit transmission and specifies the properties of both drivers and receivers. A driver must generate voltage levels of +5 to +15 volts for logic LOW input and -5 to -15 volts for logic HIGH input. A receiver must convert an input of +3 to +25 volts to logic LOW and an input of -3 to -25 volts to logic HIGH. With asynchronous transmission, a start bit and one or more stop bits are attached to the ends of each 8-bit character. When no characters are being typed on the screen, Kermit outputs stop bits. Thus, stop bits must be at least one clock period longer, but may extend any amount longer. Figure 2.3 shows the timing waveform associated with RS-232 data.

RS-232 transmitter and receiver exchange several handshaking signals for transferring data. Figure 2.4 shows the RS-232 connections used for the proposed implementation. For simplification purposes, all handshaking signals are looped back at the transmitter and receiver.

Voltage divider circuit

The input to the ASPI AD-16 A/D converter has a full range value of +3V to -3V p-p.The voltage levels from RS-232 are, therefore, passed through a voltage divider circuit to avoid clipping of the voltage levels.

C30 and C40 Boards

The transmitter and receiver were simulated individually on two SUN SPARC-stations. Optimized code was generated from a custom-coded block diagram representation of the system. For the transmitter, the generated code was loaded via ethernet on to a Banshee C30 board on a PC, where it was compiled, linked and assembled. For the receiver, the generated code was compiled, linked and assembled on the host workstation and the assembled code was loaded on to the Spectrum DBV44 board via the VMEbus.

2.2 Transmitter Model

The transmitter system of the digital modem under consideration is illustarted in Figure 2.5. The actual JTIDS system uses 5-32 CCSK encoding. Implementation of this block in SPW implies that the sampling rates at the input and output to the block should be the same as SPW3.0 does not support multirate sampling. Thus, to maintain integer number of samples per bit, the CCSK output number of bits should be an integral multiple of the input number of RS232 bits. Hence, 4-32 CCSK conversion is used so that for the case of 8 samples per bit, each codeword consists of 128 samples while each input RS232 bit is made up of 32 samples. The encoder takes 4 input RS232 bits, appends a 0 in the MSB and performs 5-32 conversion. The encoder provides error correction and spreading of the spectrum. A 32-bit buffer is used to delay the data bits from the CCSK encoder until the preamble sequence that is initially stored in a buffer is sent out. The preamble sequence is used for bit and frame synchronization and is transmitted at the beginning of each packet of data. The transmission is, thus, asynchronous with the packet size being a parameter in terms of number of CCSK levels per packet. The out-coming bits are then split into I and Q rails that are encoded using the NRZ format. The I and Q bit streams are MSK modulated and translated to an intermediate frequency f_{IF} by a cosine and a sine carrier respectively. The I and Q rails are then added and using a DAC the modulated bit stream is transmitted through the channel.

2.3 Receiver Model

Figure 2.6 depicts the receiver structure. The incoming MSK waveform is received through a channel and sampled by an ADC. An energy detector system is used at the front end of the receiver in order to detect the presence of signal. In the absence of signal only the energy detector block is operational. After the signal is detected, the synchronizer, demodulator and frame synchronizer start processing samples. The synchronizer block exploits the self synchronizing property of MSK to extract the carrier and timing information. The intermediate frequency carriers are used to translate the MSK signal to baseband. The demodulator correlates a window of one symbol period of the in-phase signal with a half sine-wave pulse to produce the value for the in-phase bit and a window of one symbol period of the quadrature signal half a symbol period later with a half sine-wave pulse to produce the value for the quadrature bit. The timing signals produced by the synchronization block determine the beginning of the correlation win-

dows. After demodulation the bits in the I and Q rails are multiplexed into a common bit stream and when the 32-bit frame sequence is detected in the incoming bits frame synchronization is achieved. The bits following the frame sequence constitute 32-bit CCSK codewords that are decoded to recover the original information bit sequence. This process of frame synchronization and CCSK decoding is done for each packet. The CCSK codewords are chosen to produce low cross-correlation values. This sys tem is capable of correcting up to 13 bit errors in each 32-bit codeword. The decoded bits are then sent to the receiver PC through RS232 for display of text on screen.

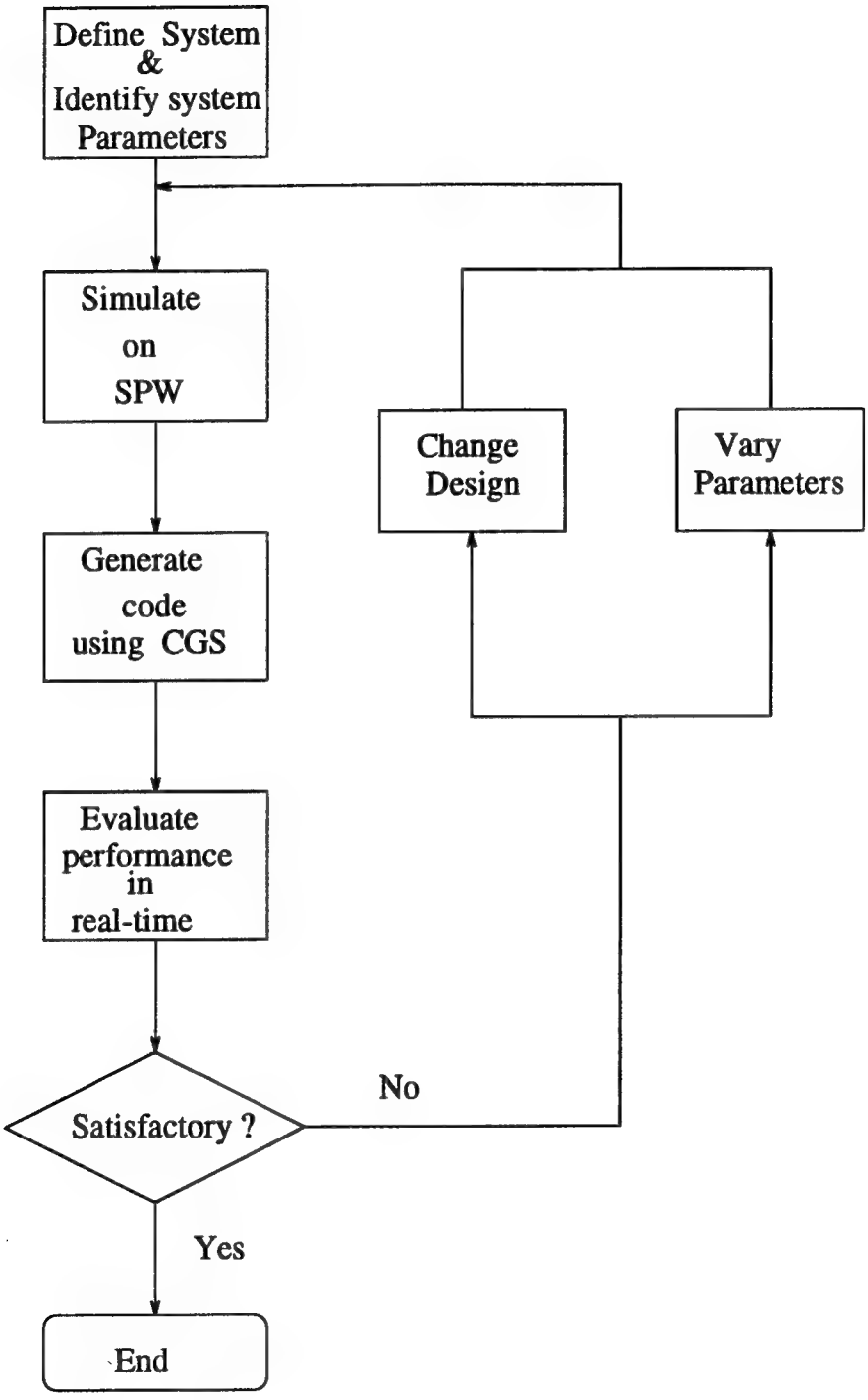
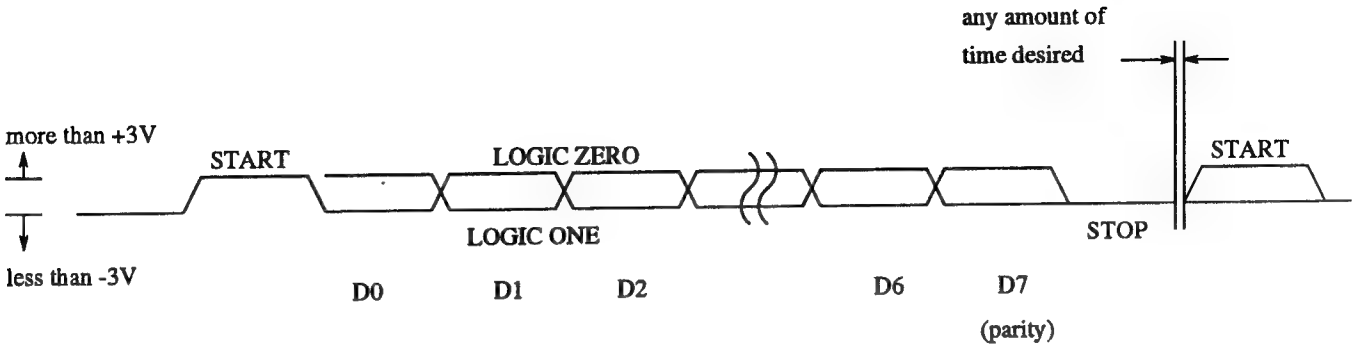


Figure 2.2: Implementation methodology for Rapid Prototyping



START and STOP bits follow positive logic

Data bits follow negative logic

Figure 2.3: RS-232 serial data timing waveform

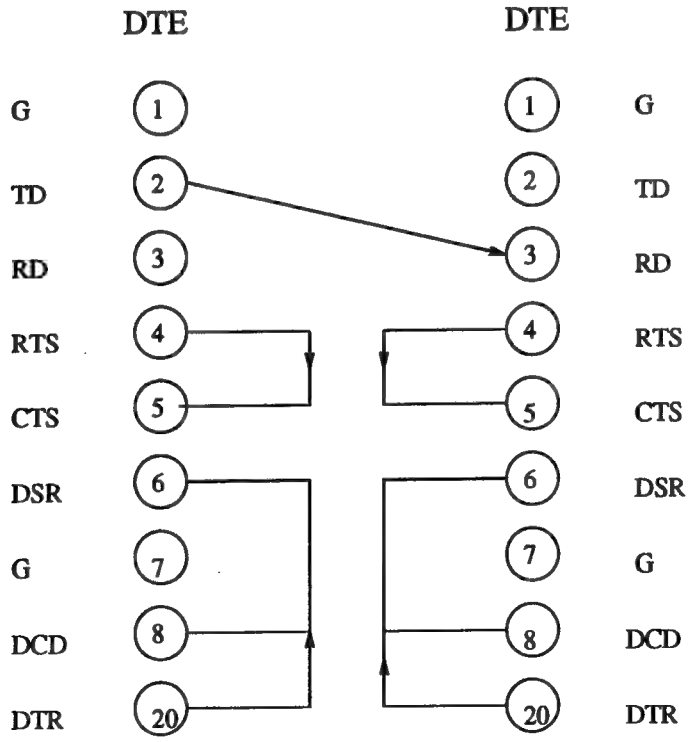
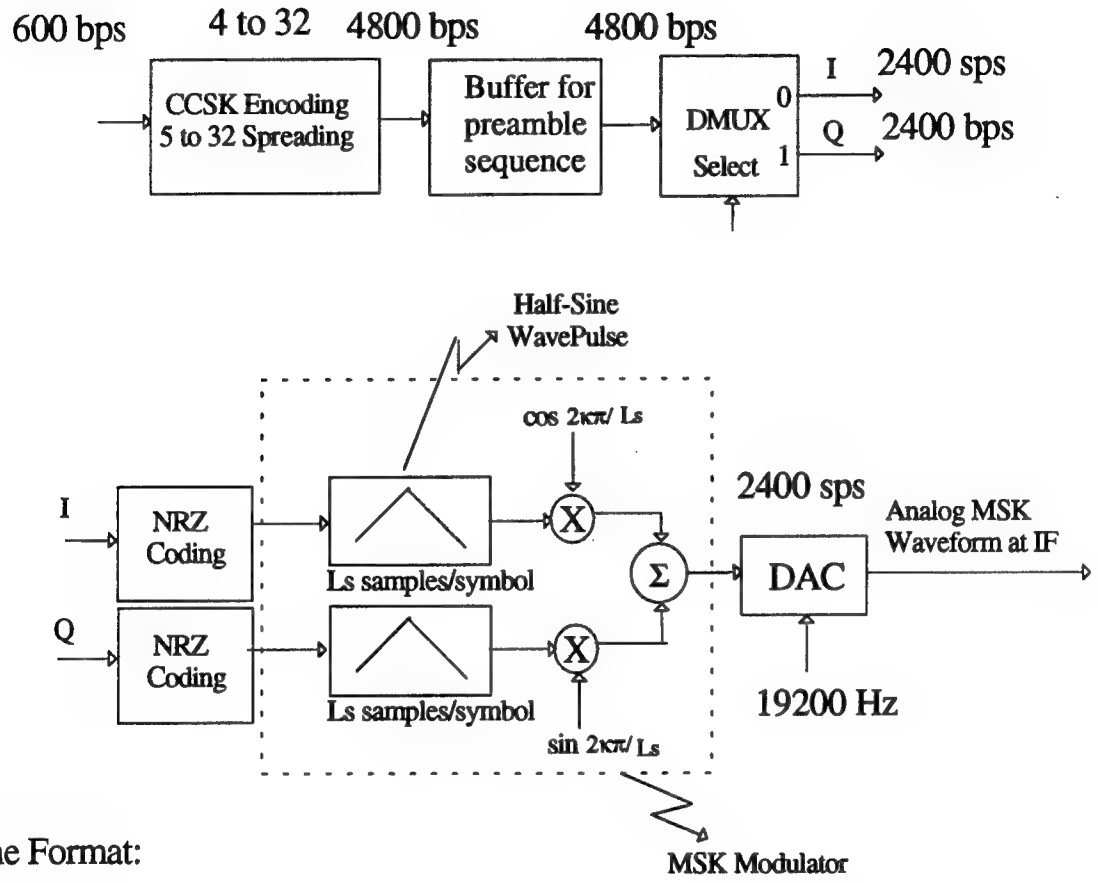
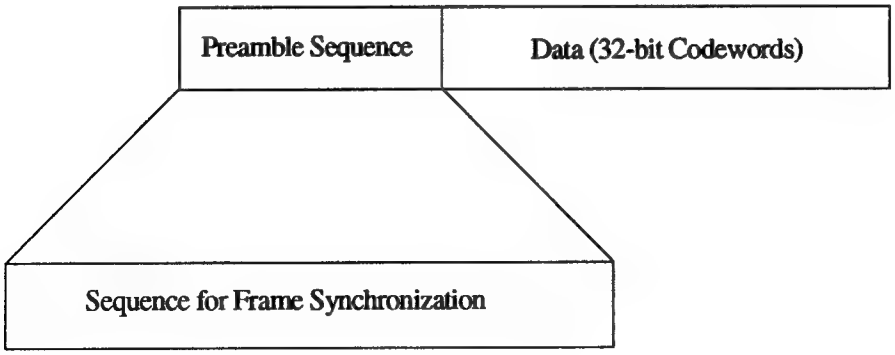


Figure 2.4: RS-232 cable connections used



Frame Format:



*

Figure 2.5: Transmitter Model

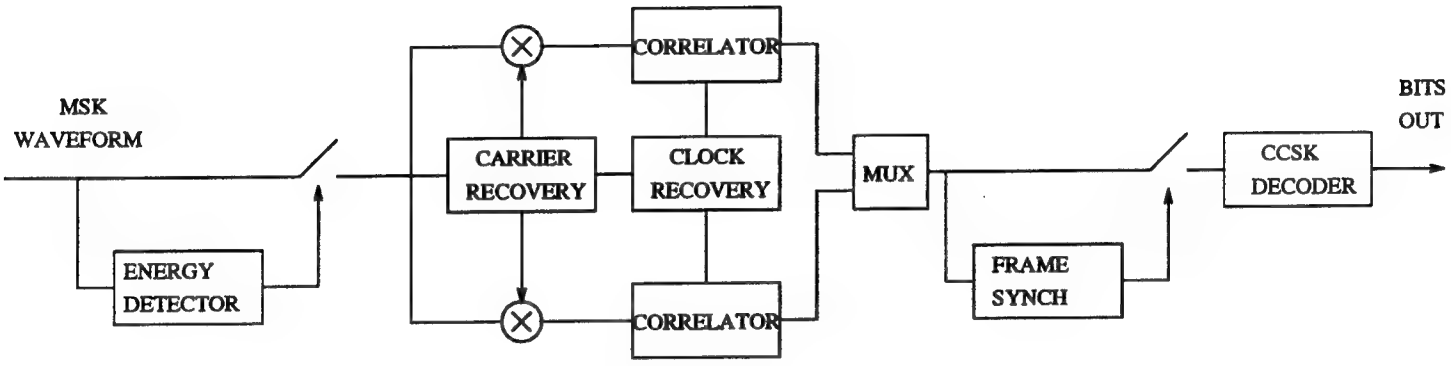


Figure 2.6: Receiver model

Optimization

3.1 Need for optimization

Data within a DSP is represented by discrete time samples. Once the sampling rate of the system is fixed, the time between samples available for computation gets fixed. It is, thus, necessary that the computations be so optimized that the time required for worst case number of computations between samples is less than the time between successive samples. This calls for optimization at the algorithmic and code levels. Optimization at the algorithmic level refers to finding ways of reducing the number of computations in the algorithms per se. This may be done through analysis specific to the system at hand. For example, simple mathematical analysis for the conventional MSK self-synchronization algorithm for the specific case of sampled data and IF rate = symbol rate led to the implementation of a computationally efficient joint carrier and symbol synchronization for the proposed system. Optimization at the code level refers to the efficient implementation of the algorithms as custom-coded blocks by identifying computational bottlenecks and exploiting the capabilities of C language for fast, bit-level implementations.

3.1.1 Algorithmic optimization

The following optimization techniques were used at the algorithmic level for the system under consideration :

- The IF rate was chosen to be equal to the symbol rate. This has the advantage that for each symbol interval, the same sample values for sine and cosine need to be generated. Further optimization was used by using the minimum possible number of samples/symbol i.e. 8 samples/symbol. 4 samples/symbol at the receiver could not be used because at this rate, the symbol synchronization algorithm fails. Another advantage is that the beginning of the symbol is an indication of the phase of the carrier. Thus, symbol timing estimation may be done jointly with carrier recovery, by hard-limiting the carrier.
- The IF rate being equal to symbol rate and the system being a sampled data system resulted in the conventional MSK self-synchronization algorithm now requiring only a squarer and one band-pass filter. This simplification resulted in a lot of computational savings.

- At the receiver, for demodulation of bits, correlate and dump was preferred to matched filtering. Matched filtering requires L_s multiply and accumulate operations per sample whereas correlation requires only 1 multiply and accumulate operation per sample, where L_s is the number of samples per symbol.
- The receiver algorithm on the whole was sped up by spreading the computations over the duration of the bit rather than doing all at the last sample of a bit. For example, the CCSK decoder takes a 32-bit received codeword and correlates it with all the entries in the CCSK look-up table. The stored codeword which results in maximum correlation is decided on as the transmitted codeword and subsequent decoding operation is performed. For the case of 8 samples/symbol or 4 samples/bit, this was implemented as follows :
  - Each incoming bit in the received codeword is correlated with the corresponding bit in each of the 32 stored codewords. This was implemented by splitting the 32 correlation computations over the first 3 samples of the 4-sample bit as 11, 10 and 11 correlations. Moreover, there is a latency of one codeword in the decoder i.e. during the presence of the current codeword, only the correlation values with all the entries in the look-up table are computed. The decision regarding the maximum of the computed correlation values is made during the next codeword. To do this, the decoder maintains a buffer, two codewords long, which stores the previous and current correlation values. For the duration of a bit, each of these blocks computes a partial correlation of the received bit and it also computes the maximum of the correlation for the previously received code word till the corresponding bit position. For example, if the decoder is computing the correlation for the 8th bit in a code word, it also computes the maximum of the correlation of the previously received codeword till the 8th bit position in the buffer. A similar technique is used in the frame synchronizer block also.
- A considerable speedup is obtained by not operating all the blocks, all the time. For example, the frame synchronizer at the receiver is operational only until frame synchronization is achieved. The CCSK decoder is not operational during this time. Once frames are synchronized, the frame synchronizer is turned off and the CCSK decoder is turned on. This is extremely important because the speed of the receiver is determined by the worst case number of computations that each receiver component executes for a particular sample in time.

3.1.2 Code Optimization

After the algorithms used in the transmitter and receiver have been optimized, the underlying code needs to be optimized to obtain real time capability.

- This involves identifying bottlenecks like *for* loops and trying to eliminate them as far as possible.
- Computations of simple functions ,which operate over a small index, like *sine* are performed ahead of time and values stored in lookup tables.
- Other buffers, variables, etc. are also initialized ahead of time, whenever possible.
- Frequently used variables (e.g. variables used inside a *for* loop) are stored in registers.
- Conditional code generation is used to generate C code depending on which inputs are connected in a particular system.
- TMS320 Floating-Point DSP Optimizing C Compiler from Texas Instruments is used to generate efficient and compact assembly code from C source. The compiler package includes an optimization program that improves the speed and reduces the size of C programs by doing such things as simplifying loops, rearranging statements and expressions, and allocating variables into registers.

Analysis and Results

4.1 Bit Rate Determination

In order to maximize the bit rate of the system the number of computations per bit need to be decreased. Algorithms that operate on bits (CCSK decoder and frame synchronizer) require a fixed number of computations per bit and thus the number of samples per bit should not affect their computational complexity . On the other hand, algorithms that operate on samples (MSK demodulator, carrier recovery, symbol synchronizer I/O processing and energy detector) require a fixed number of computations per sample and thus the number of computations per would increase linearly with the number of samples per bit. Below we derive an expression of the bit-rate for the receiver as a function of the number of samples per symbol. The number of cycles per sample required for the execution of each algorithm for the case of 8 samples per symbol (4 samples per bit) is given in the results section.

Energy Detector	58 cycles per sample
Carrier Rec & Sync	176 cycles per sample
Demodulator	142 cycles per sample
I/O Processing	64 cycles per sample
Frame Synchronizer	338 cycles per sample
CCSK Decoder	338 cycles per sample

Before frame synchronization is achieved the CCSK decoder does not operate and after frame synchronization is achieved the frame synchronizer does not operate. Thus, at any given time either the CCSK decoder or the frame synchronizer operate and hence only the computations of one of them should be taken into account. The CCSK decoder requires 338 cycles/sample when 4 samples per bit are used. Because of clock inaccuracies the number of samples per bit will not always be 4 it can be 3, 4 or 5 and thus,all the computations need to be done within 3 samples. Thus, the number of cycles per bit is

$$338(\text{cycles/sample}) * 3(\text{samples/bit}) = 1014(\text{cycles/bit})$$

In general if we have Ls samples per symbol the number of cycles per bit are divided equally among $Ls/2 - 1$ samples because $Ls/2 - 1$ is the minimum

number of samples per bit we could expect. On the average, however, each bit duration will be equal to $Ls/2$ samples. Therefore, the bit duration is given by

$$\frac{1014}{\frac{Ls}{2}-1}(\text{cycles/sample}) * \frac{Ls}{2}(\text{samples/bit}) = 1014 * \frac{Ls}{Ls-2}$$

$$\text{CCSK Decoder bit duration} = 1014 * \frac{Ls}{Ls-2}(\text{cycles/bit})$$

It is essential to note that the speed of the algorithms that operate on bits depends on the number of samples per symbol because the sampling rate of the A/D cannot be exactly equal to an integer multiple of the symbol rate of the received signal.

The number of cycles required for the other algorithms when Ls samples per symbol are used is

$$(58+176+142+64)(\text{cycles/sample}) * \frac{Ls}{2}(\text{samples/bit}) = 220 * Ls(\text{cycles/bit})$$

$$\text{Receiver Bit Duration} = 1014 * \frac{Ls}{Ls-2} + 220 * Ls(\text{cycles/bit})$$

Because these timing measurements were obtained on the C40 that has a cycle duration of 50 nsec then

$$\text{Receiver Bit Duration} = (1014 * \frac{Ls}{Ls-2} + 220 * Ls) * 50\text{nsec}$$

$$\text{Receiver Bit Rate} = \frac{\frac{2*10^7}{Ls}}{\frac{1014}{Ls-2}+220}\text{bps}$$

Due to the CCSK encoding (4 to 32 spreading) the bit rate decreases by a factor of 8 and thus the information bit rate is

$$\text{Information Bit Rate} = \frac{\frac{2.5*10^6}{Ls}}{\frac{1014}{Ls-2}+220}\text{bps}$$

For Ls=8,

Receiver Bit Rate=6427 bps

Information Bit Rate=804 bps

4.2 MSK Carrier and Timing Recovery

Both carrier recovery and symbol synchronization are derived from the received MSK signal. The MSK signal has been implemented in the form of an OQPSK with half-sine waves used as shaping pulses. The expression for the transmitted signal is

$$s(t) = A[a_I(t)|\sin(\pi\frac{t}{T_0})|\cos(2\pi f_{IF}t) + a_Q(t-\frac{T_0}{2})|\sin(\frac{\pi(t-\frac{T_0}{2}}{T_0})|\sin(2\pi f_{if}t)]$$

$$s(t) = A[a_I(t)|\sin(\pi\frac{t}{T_0})|\cos(2\pi f_{IF}t) + a_Q(t-\frac{T_0}{2})|\cos(\frac{\pi t}{T_0})|\sin(2\pi f_{IF}t)]$$

where $a_I(t)$ and $a_Q(t-\frac{T_0}{2})$ are the inphase and quadrature bit streams respectively, A is the amplitude of the MSK signal, T_0 is the symbol duration which is equal to twice the bit duration and f_{IF} is the intermediate frequency which is equal to the symbo rate $\frac{1}{T_0}$. Note that the quadrature bit stream is delayed by half a symbol in order to produce an OQPSK signal. Squaring the MSK signal will produce

$$s^2(t) = A^2[-\frac{1}{4}\cos(2\pi[2f_{IF}+\frac{1}{T_0}]t) - \frac{1}{4}\cos(2\pi[2f_{IF}-\frac{1}{T_0}]t)$$
$$+\frac{1}{2} + \frac{1}{2}a_I(t)a_Q(t-\frac{T_0}{2})|\sin(2\pi\frac{t}{T_0})|\sin(4\pi f_{IF}t)]$$

If we use a bandpass filter centered at $2f_{IF} - 1/T_0$, we can extract

$$-\frac{A^2}{4}\cos(2\pi[2f_{IF}-\frac{1}{T_0}]t) = -\frac{A^2}{4}\cos(2\pi f_{IF}t)$$

since $f_{IF} = \frac{1}{T_0}$. If we multiply the above signal with -1 then we can obtain the inphase carrier scaled by $\frac{A^2}{4}$. Note that the above derivation is presented in the continuous time case. Nonetheless, the discrete time case produces the same result when the continuous variable t is replaced by the discrete time variable t_k. If the MSK signal is sampled with a sampling frequency of $f_s = \frac{1}{T_s}$ and at time instances $t_k = kT_s + t_0^2$, then the recovered carrier signal is given by

$$\frac{A^2}{4}\cos(2\pi f_{IF}[kT_s + t_0]) = \frac{A^2}{4}\cos(2\pi f_{IF}kT_s + \phi)$$

where $\phi = 2\pi f_{IF}t_0$. The quadrature carrier can be extracted from the inphase carrier using either a delay block or a discrete derivative.

4.2.1 Delay Block

Let $x(k) = A\cos(\omega_0 k)$. Hence

$$x(k-k_0) = A\cos(\omega_0 k - \omega_0 k_0)$$

Since we want to generate a sine carrier from a cosine carrier, the integer delay k_0 has to produce a phase of $\frac{\pi}{2}$. That is,

$$\omega_0 k_0 = \frac{\pi}{2} \Rightarrow \omega_0 = \frac{\pi}{2k_0}$$
$$\omega_0 = \frac{\pi}{2}, \frac{\pi}{4}, \frac{\pi}{6}, \frac{\pi}{8}, \cdots$$

In case of the recovered carrier in the MSK receiver

$$\omega_0 = 2\pi f_{IF} T_s = 2\pi \frac{T_s}{T_0} = \frac{\pi}{2k_0}$$
$$T_0 = 4k_0 T_s$$

Therefore, in order to obtain a phase shift of exactly $\frac{\pi}{2}$ using a simple delay block, the sampling rate should be chosen such that the number of samples/symbol is 4,8,12,...samples/symbol. If the above condition is not satisfied and the delay block (see Figure 4.1) is to be used then k_0 should be chosen such that

$$k_0 = \text{round}(\frac{\pi}{2\omega_0}) = \text{round}(\frac{T_0}{4T_s})$$

which is the closest integer that produces the closest phase value to $\frac{\pi}{2}$. The absolute phase error that the delay block would produce in that case is

$$\phi_e = \omega_0 |\frac{\pi}{2\omega_0} - \text{round}(\frac{\pi}{2\omega_0})| = \omega_0 |\frac{T_0}{4T_s} - \text{round}(\frac{T_0}{4T_s})|$$
$$\phi_e = \frac{2\pi f_{IF}}{f_s} |\frac{f_s}{4f_{IF}} - \text{round}(\frac{f_s}{4f_{IF}})| = \frac{\pi}{2} |1 - \frac{4f_{IF}}{f_s} \text{round}(\frac{f_s}{4f_{IF}})|$$
$$\max\phi_e = \frac{\omega_0}{2} = \pi \frac{T_s}{T_0}$$

that would occur for $\frac{T_0}{4T_s} = 0.5, 1.5, 2.5, 3.5, \ldots$
Example :

In our system the receiver sampling rate was set to $f_s = 9638.55Hz$ while the symbol rate was equal to $f_{IF} = 2400$ symbols per sec. The phase error for this case is

$$\phi_e = 90° \frac{38Hz}{9638.55Hz} = 0.35°$$

Such a small value of the error is very insignificant and thus the delay block can be used to extract the quadrature carrier very accurately.

4.2.2 Derivative Block

The derivative of a discrete sequence is proportional to $x(k) - x(k-2)$. Since $x(k) = Acos(\omega_0 k)$ and the derivative of a cosine is proportional to the negative of a sine, then

$$x(k) - x(k-2) = \cos(\omega_0 k) - \cos(\omega_0 k - 2\omega_0) = -2\sin(\omega_0)\sin(\omega_0(k-1))$$

Note that using the derivative block we can obtain the sine carrier delayed by one sample. If we delay the cosine carrier and the MSK signal by one sample then all the signals will be in phase. It is important to note that the derivative block will always produce a sine carrier from a cosine carrier for $0 < \omega_0 < \pi$,f or all frequencies that are sampled with a sampling frequency greater than the Nyquist rate, $f_s > 2f_{IF}$. Figure 4.2 illustrates how the sine carrier can be generated.

4.3 Choosing the Bandpass filter

The bandpass filter was chosen to isolate the desired carrier. The bandpass filter,thus, needs to have a zero phase shift at the carrier frequency f_{IF} and a narrow enough bandwidth so that the other components produced as a result of squaring the MSK signal can be sufficiently removed. The major component that needs to be filtered is the cross product of the inphase and quadrature components of the MSK signal, i.e.

$$\frac{A^2}{2} a_I(t) a_Q(t - \frac{T_0}{2}) |\sin(2\pi \frac{t}{T_0})| \sin(4\pi f_{IF} t)$$

Note that the above component is a random signal centered at $2f_{IF}$, has a bit rate equal to $\frac{2}{T_0}$ and a half-sine wave duration of $\frac{T_0}{2}$ as pulse shape.

The narrower the bandwidth the less the power of the cross product and noise power that can go through. However, the receiver would require that the carrier is known more accurately so that it can fall within the bandwidth of the filter. If for example the channel causes Doppler shifts and if bandwidth becomes narrower then there is a higher chance that the carrier would fall outside the bandwidth of the filter and thus no carrier could be extracted. Also, the narrower the bandwidth the steeper the phase response of the filter and slight variations in the carrier frequency either due to doppler shifts or clock drifts would cause the phase of the carrier to change even though enough power of the carrier can be detected. To eliminate the problem of the steep phase response we can use an all- pass filter that has a phase response opposite of that of the bandpass filter.

4.4 Timing Recovery and Choice of Parameters

Because the IF carrier frequency is chosen to be equal to the symbol rate, $f_{IF} = \frac{1}{T_0}$, the clock can be extracted from the carrier. The right time to sample is the end of the symbol. The end of the inphase symbol occurs at the rising edge of the sine (quadrature) carrier and the end of the quadrature symbol occurs at the falling edge of the sine carrier. Thus, using the sine carrier as input, a rising edge and a falling edge detector, the right times to sample can be obtained. Extracting the clock from the carrier can be used in the general case where the IF frequency is an integer multiple of the symbol rate, $f_{IF} = \frac{k}{T_0}$. In that case, the right time to sample the inphase component is every k rising edge instances and the right time to sample the quadrature is every k falling edge instances. Therefore, choosing $f_{IF} = \frac{k}{T_0}$, recovering the carrier implies recovering the symbol timing since the end of the symbol (right time to sample) corresponds to the same carrier phase for all the symbols.

Moreover, in order to remove the high frequency components in the inphase and quadrature components generated by multiplying the MSK signal with the inphase and quadrature carriers respectively, $f_{IF} \gg \frac{k}{T_0}$ or $f_{IF} = \frac{k}{T_0}$ must hold. If the first case is chosen then that would require that the sampling frequency is very large and that would limit the number of computations between samples. However, choosing the second option would allow us to remove the undesired high frequency components with a small value of the carrier frequency. The smaller the carrier frequency the lower the sampling

rate required and thus the greater the available time between samples for the processor to perform computations. The smallest carrier frequency that can be chosen is $f_{IF} = \frac{1}{T_0}$. We have seen that this choice reduces the carrier recovery algorithm to single squaring and bandpass filtering.

Choosing the sampling rate to be an integer multiple of the carrier frequency and an integer multiple of the symbol rate simplifies the implementation of the transmitter. An integer number of samples per cycle allows us to generate the same valuesof the carrier for each cycle. Therefore, the numerical values of the carrier can be generated in the initialization section, stored in a look up table and used during the run time. Computing the numerical values for the carriers would require computation of sine and cosine functions that are very computational intensive tasks. The same applies for the choice of an integer number of samples per symbol.

Another important issue is the clock accuracy. The system has been implemented so that the number of samples per symbol, $\frac{f_s}{f_{IF}}$, can vary in the range $L_s - 1 \leq \frac{f_s}{f_{IF}} \leq L_s + 1$, where L_s is the expected number of samples per symbol. This wide range of values of the receiver A/D sampling frequency gives the receiver a high tolerance to clock drifts.

4.5 Results

Table 4.1: Timing Details for the Transmitter

System Component	*'C30 Instruction Cycles*
Transmitter	**346**
CCSK Encoder	96
MSK Modulator	60
IF Up-conversion	52
I/O	70

Table 4.2: Timing Details for the Receiver

System Component	*'C40 Instruction Cycles*
Receiver	**652**
Energy Detector	58
Carrier Rec & Sync	176
Demodulator	142
Frame Sync	338
Decoder	338
I/O	64

Table 4.3: BER performance of ideal and simulated systems

SNR in dB	*Ideal* P_e	*Estimated* P_e	P_i
2.1484	10^{-1}	$1.64*10^{-1.0}$	$1.22*10^{-4.0}$
5.3782	$10^{-1.5}$	$1.59*10^{-1.5}$	$6.42*10^{-11.0}$
7.3400	$10^{-2.0}$	$1.71*10^{-2.0}$	$3.23*10^{-17.0}$
8.7274	$10^{-2.5}$	$1.93*10^{-2.5}$	$2.20*10^{-23.0}$
9.8000	$10^{-3.0}$	$2.04*10^{-3.0}$	$5.10*10^{-30.0}$

$P_e \Longrightarrow$ Probability of error in bit decisions
$P_i \Longrightarrow$ Probability of error in information bits

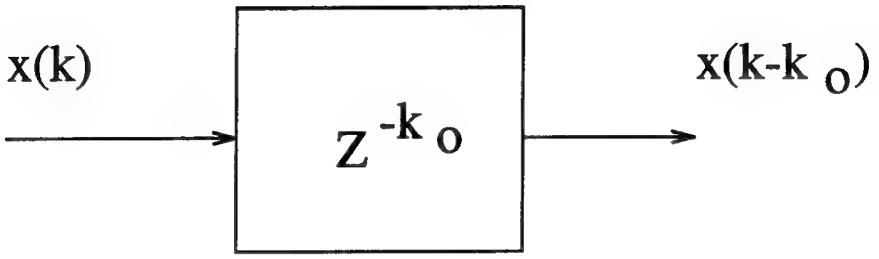


Figure 4.1: Delay Block Implementation

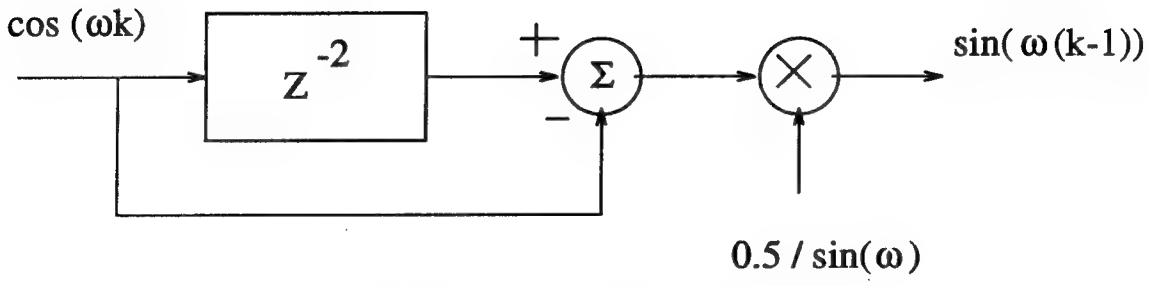


Figure 4.2: Derivative Block Implementation

Multiprox

Multiprox is an optional software product from the Alta Group (formerly Comdisco Systems Inc.) that works seamlessly with SPW to provide a distributed processing environment. Though several types of processors are supported by Multiprox, only parallel processing with the C40 will be investigated. The C40 is very well suited for such applications since it was designed with distributed processing in mind. Some of the key features of the C40 are outlined in a later section. One should expect to see a very good utilization of the processors in most distributed algorithms since the interprocessor communication(ipc) overhead between C40s can be made very small. However this is not the case with Multiprox and in many cases opting to run the code in parallel on several C40s can actually slow down the throughput compared to a single C40. For the class of low to medium complexity algorithms we observed anywhere from 20 times slower to no speed up at all. Clearly this is not what the designers of Multiprox had in mind and it could initially be frustrating. The problem lies with the way Multiprox does the ipc for the C40 processors. We suggest two alternate techniques for handling the ipc. Both the methods yield speed ups (compared to a single C40) that are unparalleled by any other method.

Using Multiprox with the SPARC processors on the other hand, it has been observed that the ethernet overhead is so high that regardless of the number of processors used, the distributed algorithm will execute many times slower (typically 10 to 100 times slower) than a single processor. Thus any attempt to alleviate the computational burden on a single processor ends in total failure if one uses SPARC processors over the ethernet. Hence our main interest in this Chapter lies in evaluating and speeding up Multiprox on the C40.

The rest of this Chapter is organized as follows. In the first Section, an introduction to Multiprox and the routine steps followed in running the software is provided. In Section 5.2 an outline of the actual interprocessor communication functions (low level) used by the standard Multiprox is provided. The reasons for the poor efficiency of the algorithms when applied in low complexity systems are also mentioned. Section 5.3 details the two alternate algorithms suggested by us. Finally Section 5.4 compares the performance of our methods with the standard Multiprox and single C40 and brings out the advantage of using our techniques.

5.1 Standard Multiprox

Multiprox is a fairly complex software and using the tool needs many interactions from the user. This Section begins with an overview of Multiprox and then outlines the steps involved in using the software.

Standard Multiprox is a multiprocessor code development software that allows the user to run a block oriented system in SPW, on several processors. The processors used have to be of the same kind. For example one cannot use a C40 along with a SPARC processor. One can put the software in perspective and its interactions with the other software products of SPW as shown in Figure 5.1.

Once a block diagram has been created, the user has to launch a Multiprox editor from the block diagram editor. A partition of the designed system needs to be specified. For optimum performance, the partition has to be carefully selected so that the ipc per iteration is minimized and at the same time the processors are more or less equally loaded. This requires some practice and many times it also could be time consuming. Many block diagrams simply cannot be partitioned so that the code is equally distributed among all the processors. In any case, once a satisfactory partition has been achieved, the user has to edit the "region parameters" file for each partition. The file consists of the "processor value" and "processor type" which target the specific processor (in a multiprocessor environment) and the type of the processor. Typically for C40s, the first parameter takes numbers 1, 2, 3, 4 and so on while the processor type would be C40. The exact details of this depend on the software that actually interacts with the processors (mostly designed by the board vendors). After these parameters have been correctly specified, the user needs to specify an "Architecture Description File" (ADF for short) which contains the description of the multiprocessor set up and also specifies exactly which ipc blocks are to be used for interprocessor communications. After verifying (or creating) the ADF file, the software is ready to create, compile and run the partitions on the respective processors.

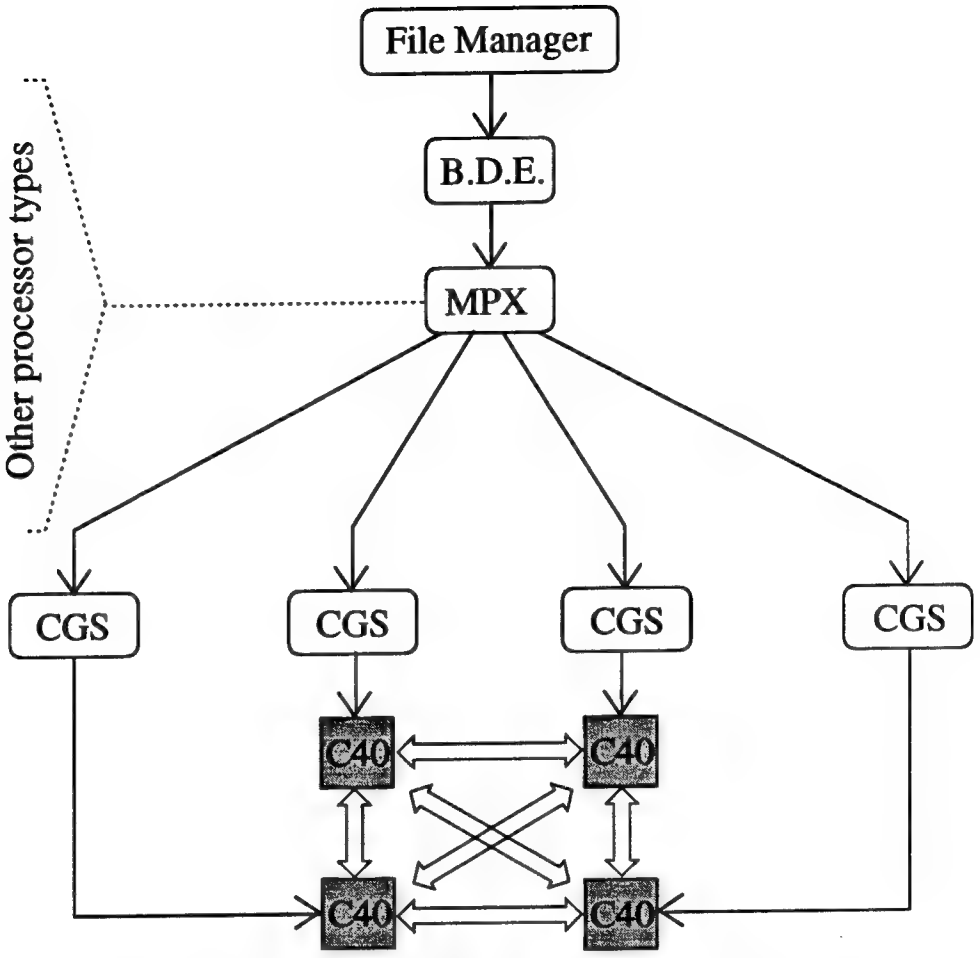


Figure 5.1: Multiprox and its interactions with the rest of SPW

5.2 IPC in Standard Multiprox

Standard Multiprox (abbreviated to S-MPX) inserts an input and an output block at each connection between partitions. These blocks (which are hidden from the block diagram) are used to specify the exact ipc method.

As an initialization step, S-MPX establishes logical (software) links between processors using four functions, *fifoInitX()* and *fifoConnectX()* where *X* could be *Input* or *Output*. The *fifoInit* functions are called during the initialization stage. The functions establish the actual physical connections that exist and also set up the software or logical links for each signal that runs accross a partition. They also do an initial synchronization by a fairly

complicated procedure of exchanging tokens, the details of which may be found in the *CGS/Multiprox Interface Porting Kit* that is distributed by the Alta Group of Cadence Systems Inc. The *fifoConnect* functions are used to synchronize the processors at the beginning of the run stage. This level of synchronization assures that all the processors are actually at the beginning of the run stage. All of these functions are of major complexity and typically take an order of 10^5 cycles.

The actual ipc occurs via a call to either *fifoRead()* or *fifoWrite()* function. Both the functions directly access the communication ports of the C40 (without any DMA intervention). The processor CPU will only proceed after the data has been properly exchanged. It is to our understanding that the reason for a high degree of synchronization performed in the initial setions of the code is that the processor which receives data is ready when the transmitting processor is ready with data. In any case, the peripheral bus in the C40 (which is used to access any communication port) can be halted when the port is not ready to take the data from the processor. This also can slow down the ipc. Another major reason for the high complexity of these read/write functions is the overhead of the CPU writing each data element (if there are more than one per logical link) after the previous data is ackowledged by the port.

The efficiency of S-MPX ipc goes down dramatically if more than one logical link exists between any two partitions. While it is generally accepted that in any multiprocessor algorithm, the overall execution can only proceed as fast as the slowest (most loaded) processor, the performance of S-MPX when more than link exists is simply unacceptable in any real time system. At present the reason why the performance degrades so dramatically is not clear (see the results in Section 5.4 for verification of the previous statement).

5.3 Our IPC Methods

The key issue is how to handle the ipc so as to achieve an acceptable utilization from multiple processors. For simulation purposes, if a buffer is provided so that the ipc functions are executed for blocks of data, there is a significant improvement in the CPU utilization even for S-MPX. On the C40, the speed of execution can also be affected if one provides a buffer. In order to achieve a good throughput, the overhead of the ipc should be distributed over the entire buffer length. In a real time implementation, this is important because, suppose the ipc transfer of the buffer is done between any two samples (or iterations) of the system. Unless the A/D can interrupt

the transfer process, samples will be lost at the A/D. Interrupt driven I/O is computationally costly owing to the latency of the processor, function call overheads, etc. On the C40, the distribution of the ipc can be readily achieved using the DMA operations.

Using Multiprox, one can successfully achieve about 0.75% CPU utilization with a buffer size of 1000. However the system has to be fairly complex in order to offset the ipc overhead. Also Multiprox does not use the advantage of the C40 that is given above. As a consequence a real time system cannot be run on multiple processors using Multiprox. No meaningful throughput can be achieved through such a system. To circumvent this problem, an efficient ipc protocol was developed and is given in Section 5.3.2. In the following Section the advantages that the TMS320C40 processor provides are summarized.

5.3.1 The TMS320C40 Processor

The TMS320C40 is highly optimized for parallel processing applications.

Some of the advantages of using the TMS320C40 as a parallel processing

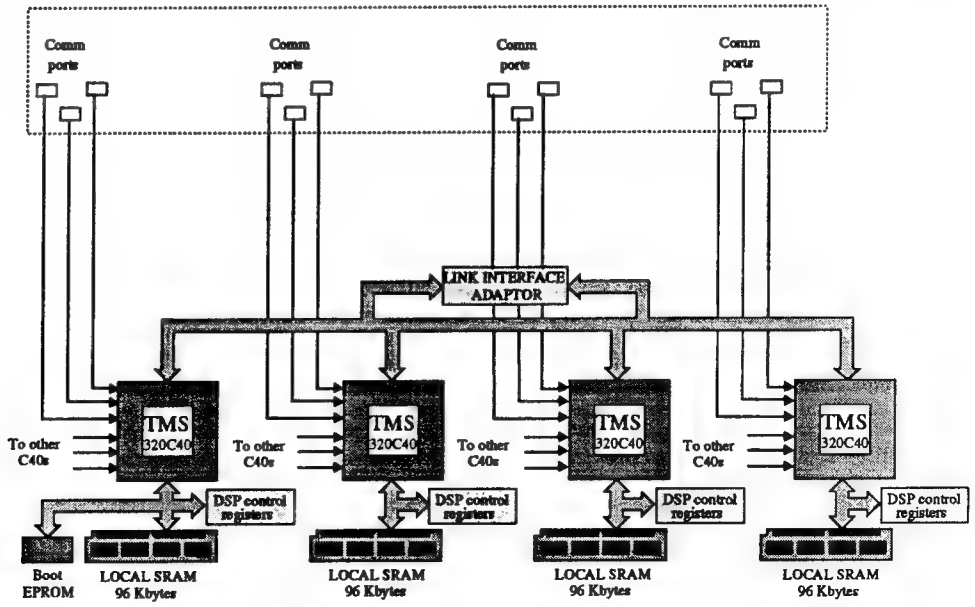


Figure 5.2: Quad C40 Board Layout

node are

- Node speed of about 50 MFLOPS.
- Single cycle (50 nanoseconds) 32 bit floating point multiplier.
- Single cycle (hardware) divide and inverse square root.
- Single cycle floating point multiply and add (parallel operations).
- Single cycle delayed branch loops.
- Six 20 Mbytes/sec comm ports for node to node communications.
- Six bidirectional DMA channels for ipc.
- Highly configurable for parallel processing topologies.
- I/O of 100 Mbytes/sec on Global bus, 100 Mbytes/sec on the local bus and 120 Mbytes/sec on the six comm ports.

5.3.2 Parallel Processing with C40

Some of the features of the C40 such as dedicated parallel communications ports (commport) and DMA channels allow us to develop an efficient protocol. Each commport is bidirectional and can be used to send and receive data simultaneously. Port arbitration is done entirely by the C40 hardware and thus it is much simpler to transfer data in both directions. The total commport transfer rate of 20 MBytes per second allows for a high degree of parallel processing. As shown in Figure 5.3, our ipc algorithm IPC-2 is based on a polling scheme where the CPU polls the appropriate DMA control register when it is ready to transmit or receive data. If the previous DMA transfer (to or from the relevant commport) was completed then the CPU writes or reads from the DMA buffer and restarts the DMA for the transfer operation. This procedure decouples the CPU from doing the memory transfers completely and also prevents the peripheral bus from getting halted. However there are exceptional cases where the appropriate flags in the DMA control register are not properly set. We observed that the combination of all the four methods (mentioned in Chapter 9 of the C40 manual) is always OK to use.

IPC-1 is very similar except that it does not rely on the DMA control register. In this case, the receiving processor automatically generates an acknowledgement that is transferred to a known memory location on the transmitting processor. All this handshaking is done by the respective DMAs and the CPU is not bothered at all. When the C40 needs to send or receive data, it checks the memory location for an acknowledgement. While this method is definitely of greater appeal than IPC-2, it also takes slightly longer owing to all the additional DMA transfers (for the DMA has to reinitialize its direction of transfer etc.).

The major advantage of IPC-1 is that in systems with feedback between

the two processors, the feedback signal can very easily be piggybacked on the acknowledgement link. As shown in the results the penalty for an extra data transfer is 2 cycles (or 100 nanoseconds) per iteration which is truly negligible.

In most signal processing applications, data transfers between the processors occur in a deterministic order and the transfers are done periodically. We make effective use of this simplicity by constructing a linked list of DMA autoinitialization sequences as shown in Figure 5.4. Each linked list element consists of seven fields (clarity permits only a couple of relevant ones to be shown). The head and tail of the linked list are used to signify a halt code which enables any C40 in the network to halt all the other processors if necessary. The rest of the elements of the list contain the addresses of the data and direction of data transfer. In Figure 5.4, the direction field is a 0 if data is transmitted and is a 1 if data is received. Further simplification can be achieved in systems where data transfer occurs in one direction only. In that case one can buffer all the data to be transferred in each iteration and do the transfer operation only once per iteration.

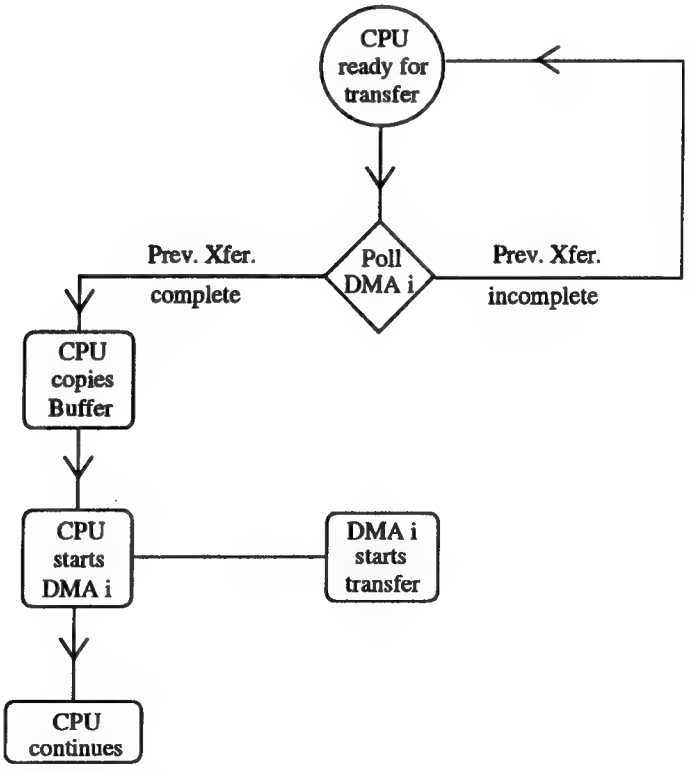


Figure 5.3: CPU-DMA Interactions for IPC

As listed in Table 5.1, IPC-1 takes 24 cycles or 1200 nanoseconds for transferring one word (4 bytes) whereas IPC-2 takes 20 cycles. There is a case of "diminishing returns" when one uses more processors. Since a single word read/write takes about 24 cycles, even in the hypothetical case where each sample (or iteration) takes the same number of cycles on each C40, the algorithm complexity cannot be less than 24 cycles.

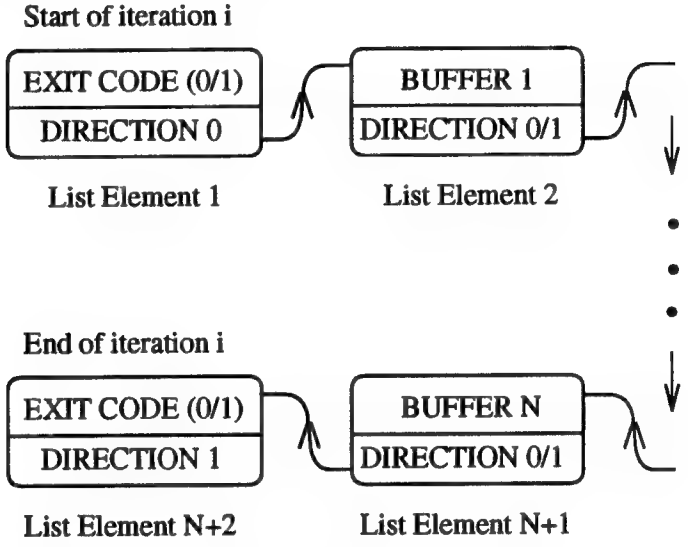


Figure 5.4: Linked List & Operation of DMA

In systems that can withstand delays (eg. those that do not have feedbacks), all ipc methods benefit to varying extents by the use of buffered transfers. The basic method is illustrated in Figure 5.5 for the cases of S-MPX and either of the methods suggested by us (IPC-1 or IPC-2). For IPC-1 and IPC-2, if the DMA is transferring the contents of buffer B_1 then at the same time, the CPU can write to another buffer B_2. When B_2 becomes full, the CPU can poll the DMA channel's control registers to determine if B_1 was successfully transferred. Then the buffers need to be switched and the processors can continue. Whereas by switching buffer pointers (which takes a couple of cycles), our methods can essentially transfer any size buffer in about 24 cycles, there is no advantage of using two buffers and switching them while using S-MPX. This is because unlike our techniques, S-MPX does not operate the DMA in parallel with the CPU. Of course we have to assume that the DMA finishes the previous transfer before the CPU is ready with the new buffer in order to realize the entire buffer transfer in 24 cycles or less using our methods. In most DSP type algorithms this will be true since the CPU will do a certain amount of processing before it is ready with the buffer.

There is also the issue of CPU and DMA conflicts when they try to access the same resources. When there is only a scalar word (4 bytes) being transfered, the maximum probability of conflict occurs when the DMA channel is autoinitializing because it involves 7 word transfers in the unified mode (see the TMS320C40 Manual for more details). Thus it is 7 times more likely to clash with the CPU when it is in its autoinitialization procedure. On the other hand for buffered transfers, CPU/DMA conflicts can occur more frequently and depending on the size of the buffer it might not be as important to consider the effect of the autoinitialization sequence itself.

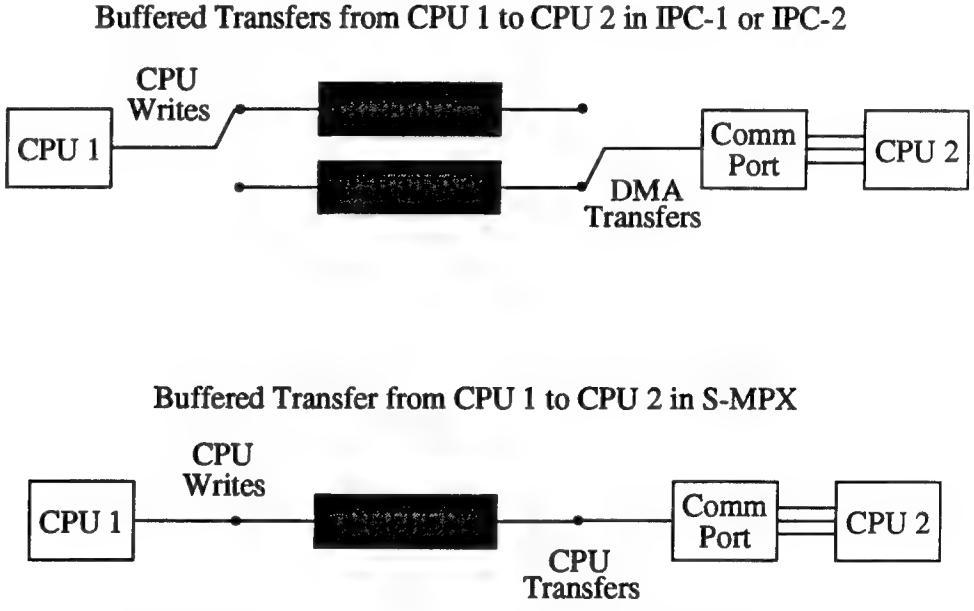


Figure 5.5: Use of buffers for IPC-1, IPC-2 and S-MPX

One important feature of the C40 that is relevant here is the concurrent operation of the DMA processor without almost any intervention from the CPU. Thus if the DMA is transferring the contents of buffer B_1 then at the same time, the CPU can write to another buffer B_2. When B_2 becomes full, the CPU can poll the DMA channel's control registers to determine if B_1 was successfully transferred. Then the buffers need to be switched and the processors can continue.

The results of buffered transfers in S-MPX is listed in Table 5.2. The system used to get the timing results was of high complexity (a few thousand cycles per iteration). Thus running a complicated algorithm on two processors using S-MPX did produce a speed up of nearly 2. Since the buffered transfer methods for IPC-1 and IPC-2 have not yet been implemented, its performance is not clear. Our prediction is that it will yield similar results.

The block descriptions for both the methods can be found in Chapter 6. All four blocks have much in common during SPW initialization. The autoinitialization sequences of the DMA are first set up for all possible operations of the particular DMA channel. This is done using the "autoinit" link structure. The DMA channel index itself is obtained from a connectivity matrix that is included in a "mytypes.h" include file. The values of the different fields of the linked list element are set up. The number of data words to be transferred depends on the number of signals that are input or output from the block. CPU accesses are given priority over DMA accesses by setting the corresponding DMA channel control register bits.

In IPC-1, the control register is also set for a write operation to the comm. port. Following the write, the DMA channel automatically autoinitializes for receiving data or acknowledgement. The write operation involves either data to be transferred (if the processor is the transmitter) or the acknowledgement corresponding to *the data received on the previous transfer* if the processor is the receiver. After the read operation (either data or acknowledgement) the DMA channel halts and waits for the CPU to restart it. For its part, the transmitting CPU first checks if the previous transfer has been acknowledged. If so it restarts the DMA for the next transfer. On the other hand, the receiving CPU checks its OK flag (the last word of the transferred data buffer). If the flag is set then it reads from the data buffer and restarts its DMA channel. In this manner the CPU is almost completely free of involvement in the memory transfers.

Using IPC-2 reduces the transfers per iteration since there are no acknowledgements involved. The transmitting C40 has its DMA autoinitialization sequence set for writing to the comm. port while the receiving DMA has its sequence set for read. When the CPU needs to transfer data, it checks whether the previous DMA transfer completed. It does this by checking the transfer counter, interrupt flag and the start bits in the DMA channel control register. The CPU proceeds only if the DMA finished the previous transfer.

5.4 Comparison of the Methods

The CPU utilization of a processor, defined as the percentage of time that the CPU actually executes the user's program, not including the interprocessor communication (ipc) overhead, is fairly good when Multiprox is used for systems that are computationally intensive. However simpler systems that take of the order of a few hundred instruction cycles do not benefit from the use of Multiprox. Since typically rapid prototyping involves quick

(real time) design prototyping which in many cases would also be reasonably of low complexity (a highly complex system is simply not suitable for rapid prototyping, atleast using limited resources !) multiprocessor routines have to work with similar low level throughputs, i.e. low complexity and yet large volume data transfering capability is essential for any interprocessor communication system.

We compared the performance of all the ipc methods suggested in this Chapter with that of using a single C40. The system that was chosen for this purpose is shown in Figure 5.6. SYSTEM-1 and SYSTEM-2 that were used to obtain the results in Table 5.1 are shown in Figures 5.7 and 5.8 respectively. The difference in the two systems was that the number of words transferred per iteration in SYSTEM-1 was 1 while it was two word transfers per iteration for SYSTEM-2. We also checked the real time capability of our algorithms against S-MPX and a single C40 (using CGS). Specifically we modeled a Delta Modulation/Demodulation system using the different ipc algorithms and CGS (on a single C40). The results are given in Table 5.2. It is clear that our methods prove advantageous at high data rates (due to the low complexity of the ipc algorithms).

All the Multiprox algorithms (including IPC-1 and IPC-2) were run on two processors. Also the test for real time capability used in Table 5.2 is that the rate of iterations (or rate of sampling) is the same as the sampling frequency set for the D/A conversion on the first processor. If the rate is slower than the sampling frequency, then the system is not real time.

System	Method used	IPC/Sample	Time for 3×10^6	Gain
SYSTEM-1	IPC-2	2	53 seconds	1.85
SYSTEM-1	IPC-1	2	54 seconds	1.82
SYSTEM-1	Standard mpx	2	181 seconds	0.55
SYSTEM-1	Single C40	2	99 seconds	1.0
SYSTEM-2	IPC-2	1	52 seconds	1.86
SYSTEM-2	IPC-1	1	54 seconds	1.83
SYSTEM-2	Standard mpx	1	84 seconds	1.16
SYSTEM-2	Single C40	1	98 seconds	1.0

Table 5.1: Timing summary of SYSTEM-1 and SYSTEM-2

Tone in	samp.Freq	Method used	IPC	Time/3×10^6	Realtime
7 kHz	67.4 kHz	IPC-2	2	45 seconds	yes
7 kHz	67.4 kHz	IPC-1	2	45 seconds	yes
7 kHz	67.4 kHz	Standard mpx	2	174 seconds	no
7 kHz	67.4 kHz	Single C40	2	99 seconds	no
3 kHz	29.6 kHz	IPC-2	2	101 seconds	yes
3 kHz	29.6 kHz	IPC-1	2	101 seconds	yes
3 kHz	29.6 kHz	Standard mpx	2	160 seconds	no
3 kHz	29.6 kHz	Single C40	2	101 seconds	yes
1 kHz	9636 Hz	IPC-2	2	311 seconds	yes
1 kHz	9636 Hz	IPC-1	2	311 seconds	yes
1 kHz	9636 Hz	Standard mpx	2	311 seconds	yes
1 kHz	9636 Hz	Single C40	2	311 seconds	yes

Table 5.2: Timing summary of the system in Figure 5.6

We also benchmarked buffered S-MPX and CGS on the SPARC on two other systems that had almost loaded both the processors equally. The results are presented in Table 5.3. The systems differed in the computational complexity. SYSTEM-1 was much more complex than SYSTEM-2. The results given are for 10^6 samples. The details of the systems used are not important here and thus are not given.

SYSTEM	Method used	Buffer Size	Time in secs
SYSTEM-1	SPARCstation II		66.0
SYSTEM-1	Single C40		35.0
SYSTEM-1	Multiprox on 2 C40s	1	19.0
SYSTEM-1	Multiprox on 2 C40s	1000	17.0
SYSTEM-2	SPARCstation II		22.0
SYSTEM-2	Single C40		28.0
SYSTEM-2	Multiprox on 2 C40s	1	32.0
SYSTEM-2	Multiprox on 2 C40s	1000	17.0

Table 5.3: Timing summary of SYSTEM-1 and SYSTEM-2

While the SPARCstation was executing the systems, about 90 to 95% of the CPU was used by the programs. One can deduce that using a large enough buffer with SPW Multiprox, there is very good processor utilization (if the computations are sufficiently intensive). However as mentioned earlier, this

cannot be used in a real time system. Also the SPARCstation outperforms the C40 when the complexity of the algorithm is very low. Typically most of the operations done in SYSTEM-2 were of the logical kind and thus the reduced architecture of the SPARCstation II processor provides advantages in such cases. However the C40 DSP processor outperforms the SPARC processor significantly when the algorithm is fairly complex, involving floating point operations.

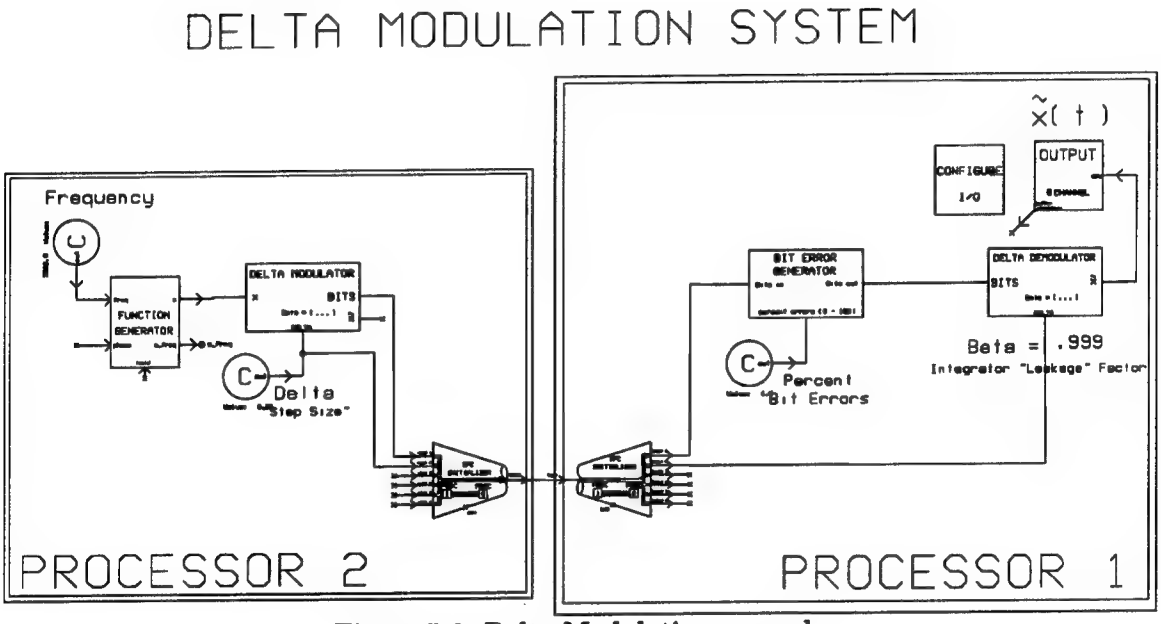


Figure 5.6: Delta Modulation example

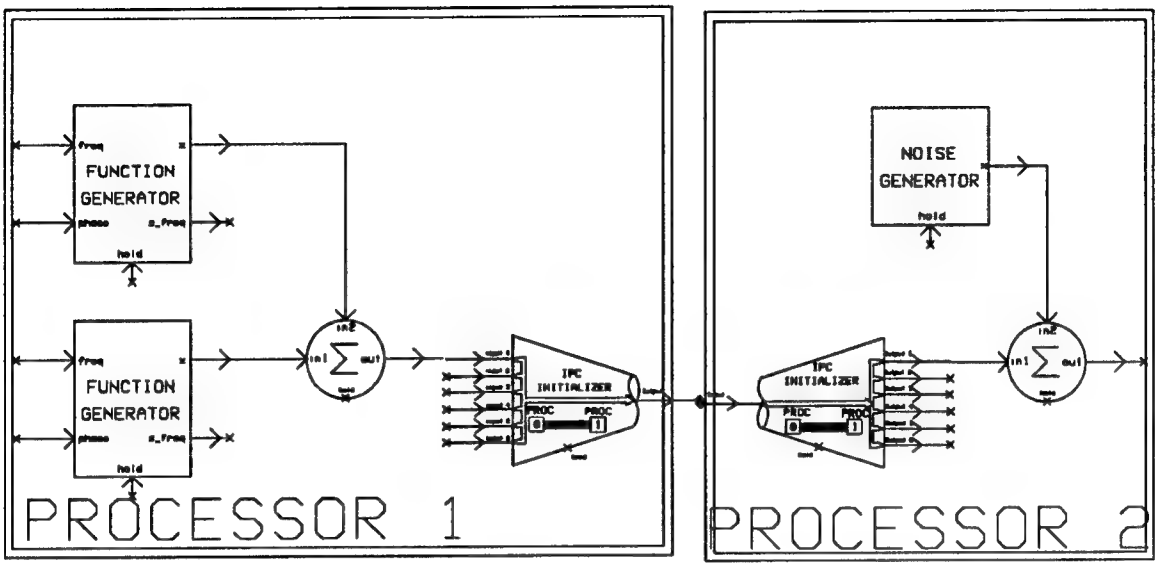


Figure 5.7: SYSTEM-1 with one data transfer used in Table 5.1

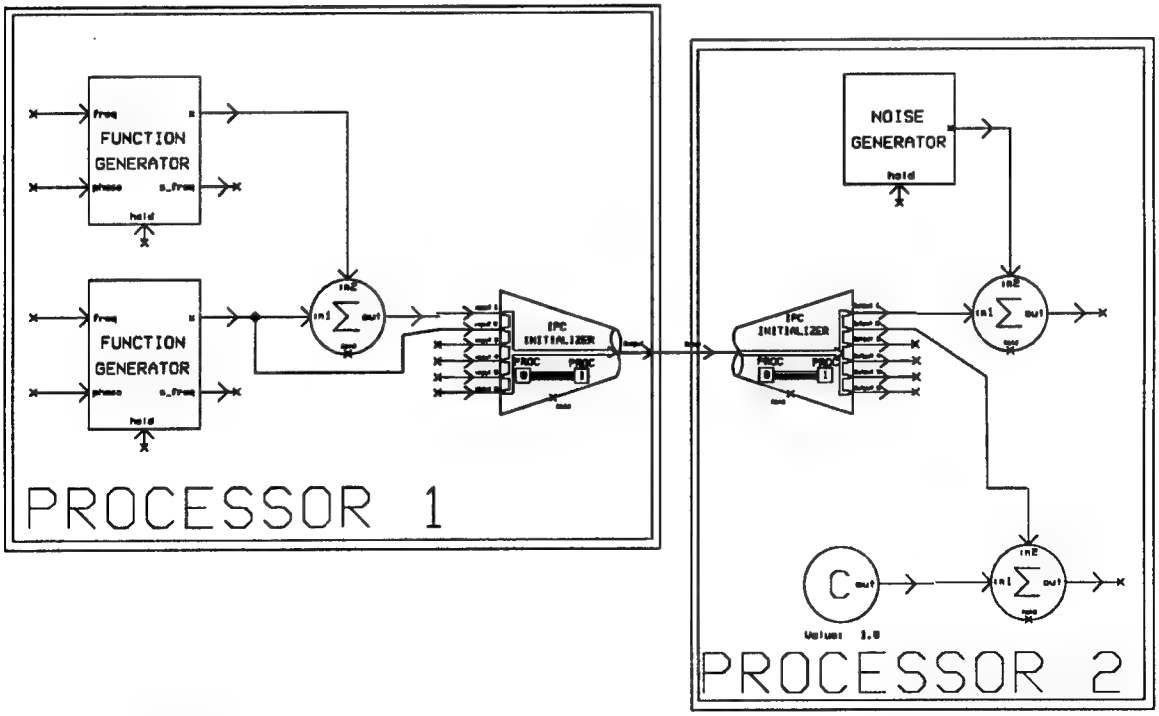
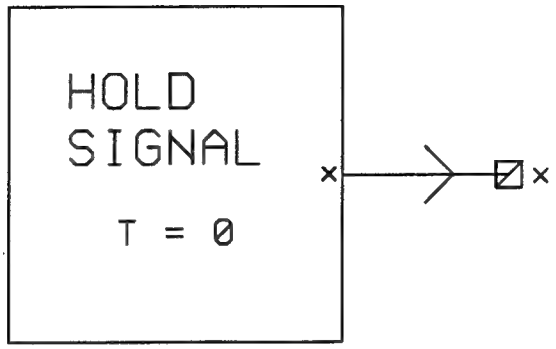


Figure 5.8: SYSTEM-2 with two data transfers used in Table 5.1

Transmitter

Name **HOLD SIGNAL** */caedata/chaoslib/hold_true*



HOLD BLOCK PARAMETERS	
Feed through type	ALL_FEED_THROUGH
Hold Period	32

Type Primitive

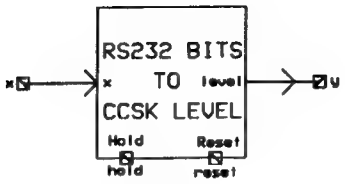
Description Initially, when the current iteration number is not equal to the iteration number parameter, this block outputs a 1. When current iteration number becomes equal to the iteration number parameter the block outputs a 0 for one sample. For the next (period - 1) samples it outputs a 1. It then repeats this stream of one 0 and (period -1) 1's till the end of simulation.

Inputs & Outputs This block has one output x.

Parameters

- **Hold period:** number of samples per period of one 0 followed by (period-1) 1's.
- **Iteration number:** sample number at which this block starts outputting a 0 followed by (period-1) 1's.

Name **RS232 BITS TO CCSK LEVEL** */caedata/chaoslib/rs232_ccsk*



RS232 TO CCSK BLOCK PARAMETERS	
Feed through type	ALL_FEED_THROUGH
Precision (number of bits)	5
Data Rate	300.0
Input Sampling Rate	900.0
Number of sync bursts per packet	6

Type Primitive

Description This block is used to convert input information bits from RS232 to CCSK levels . Characters typed on a PC screen in the Kermit environment are sent to the serial port of the PC as bits representing the characters in 7-bit ASCII notation + 0 MSB bit. Since bit transmission is asynchronous, these 8 bits representing a single character are preceded by one start bit (logic 1) and succeeded by several stop bits (logic 0) depending on the speed of typing. These bits are sent , LSB first and MSB last. Before being sampled by the A/D converter, these bits follow the logic convention as given below:
Start bit - positive voltage
Stop bits - negative voltage
Data bits - logic 0 - positive voltage
logic 1 - negative voltage
However some boards like the DB212 A/D board introduce a gain of -1 and hence the bits now follow the logic conventions as shown:
Start bit - negative voltage
Stop bit - positive voltage
Data bits - logic 0 - negative voltage
logic 1 - positive voltage
Thus , in a given bit stream, a start bit can be detected after several stop bits by looking for a falling edge. The usual CCSK encoder takes a level representing 5 information bits and outputs 32 bits. Thus, this block should

actually have taken 5 RS232 bits to form one CCSK level. However, due to certain constraints imposed by Kermit software on the permissible information bit rate, a simplifying assumption was made and the block now takes 4 bits, appends a 0 in the MSB and forms a level. A consequence of this assumption is that the CCSK encoder now gets levels corresponding to only the first 16 codewords.

This block makes an assumption that there are at least 3 stop bits between characters. When stop bits are being received, this block outputs a level 31 constantly till a falling edge is detected, indicating presence of a start bit. At this point it starts constructing 3 levels corresponding to one character using 1 start bit, 7 bits representing the character, the 0 MSB bit and 3 stop bits. In the end again, when more stop bits occur, it starts outputting level 31 till the next falling edge. If we consider the stream of level 31 (stop bits) as being a set of CCSK levels + some extra samples before a falling edge is detected, these extra samples may not be sufficient to form the last CCSK level. This problem is taken care of by the FRAMER block.

In order to incorporate asynchronous transfers as well, the block uses a circular linked list to buffer the CCSK levels while the frame sequence for synchronization is being sent. The buffer is decremented each time the required number of stop bits occur consecutively. Instead of actually recreating the list, the list element that is dropped is actually attached to a secondary linked list. When a new packet begins, the linked lists are switched and the algorithm proceeds. To this end, a third buffer actually speeds up the block processing time. When the linked list that is currently being used becomes empty, it signifies that there have been sufficient number of stop bits to account for the extra CCSK bits sent during the frame synchronization. In such an event the input RS232 bits directly get converted to CCSK levels with no extra buffer delays.

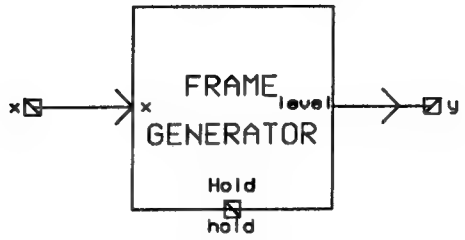
Inputs & Outputs

Input x is the RS232 bits input and output y is the CCSK levels output

Parameters

- **Data rate:** input bit rate
- **Sampling rate:** number of samples per sec.
- **Number of sync bursts per packet:** number of 32 bit frame sync sequences per packet which the sync burst generator block should send
- **precision:** not currently being used

Name **FRAME GENERATOR** */caedata/chaoslib/framer*



FRAMER BLOCK PARAMETERS	
Feed through type	ALL_FEED_THROUGH
Data Rate	300.0
Input Samping Rate	9600.0

Type Primitive

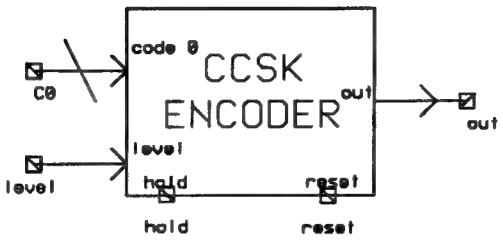
Description This block overcomes the problem which might occur if the level 31 at the output of the RS232 to CCSK level block, corresponding to stop bits, is not present for a duration equal to an integral multiple of a single CCSK level duration. For example, if bit rate = 300, sampling rate = 9600, then a a CCSK level corresponding to 4 bits exists for 4 * 9600/300 = 128 samples. But, due to the asynchronous nature of RS232 data, it is highly probable that the stream of level 31 samples might exist for a duration equal to an integral multiple of 128 + some extra samples (< 128). This block overcomes this problem by extending the number of these extra samples to 128 , while buffering the input data.

Inputs & Outputs Both input x and output y represent CCSK levels

Parameters

- **Data rate:** information bit rate of RS232 data
- **Sampling rate:** number of samples per second

Name **CCSK ENCODER** */caedata/chaoslib/encoder2*



ENCODER BLOCK PARAMETERS	
Feed through type	ALL_FEED_THROUGH
Code Size in Bits	32
Block Code of 0 (if port disconnected)	2095647468
Code Vector Size	32

Type **Primitive**

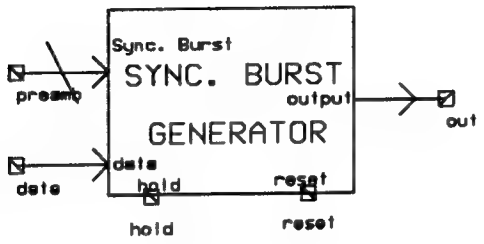
Description **This block performs 5-32 CCSK encoding. It accepts a level corresponding to 5 information bits and outputs 32 CCSK encoded bits in the duration of the level corresponding to 5 bits. This block also has an input for the user to specify the zero level CCSK codeword of his choice. Computational efficiency is improved by performing bitwise operations as much as possible.**

Inputs & Outputs **Input *level* represents CCSK level while input *C0* represents an optional user-specified zero level CCSK codeword. Output *out* corresponds to CCSK encoded bits.**

Parameters

- **Code size in bits:** size of CCSK codeword in bits
- **Block code of 0:** decimal equivalent of codeword corresponding to level 0
- **Code vector size:** size in bits of codeword vector if input port code 0 is used

Name **SYNC. BURST GENERATOR** */caedata/chaoslib/sync_gen*



SYNC. GEN BLOCK PARAMETERS	
Feed through type	ALL_FEED_THROUGH
Preamble Size in Bits	32
Preamble Sequence (if port disconnected)	268435455
Preamble Vector Size	32
Initial Output Value	1.0
# frame sync sequences	6

Type Primitive

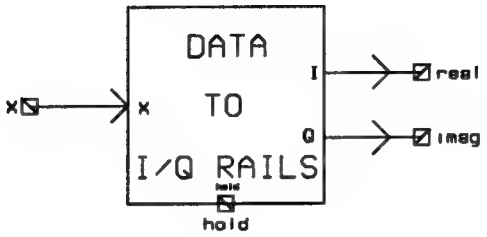
Description This block generates one or more preamble sequences to be transmitted at the beginning of each frame of CCSK encoded bits for frame synchronization.

Inputs & Outputs Input *data* represents packets of CCSK encoded data bits and input *preamb* represents an optional user-specified preamble sequence.

Parameters

- **Preamble size in bits:** size of preamble sequence in bits
- **Preamble sequence:** decimal equivalent of desired preamble sequence
- **Preamble vector size:** size in bits of preamble sequence vector if input port sync burst is used
- **Initial output value:** initial value of output during block initialization
- **Number of frame sync sequences:** number of preamble sequences to be transmitted at the beginning of each packet

Name **DATA TO I/Q RAILS** */caedata/chaoslib/data_iq*



DATA TO I/Q RAILS BLOCK PARAMETERS

MAIN PARAMETERS:

None

ALL_FEED_THROUGH

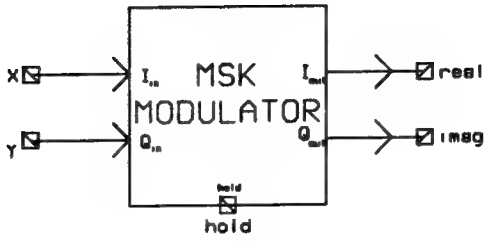
Type **Primitive**

Description **This block is used to demultiplex the input bit stream into I and Q channels. It also extends the bit duration in each channel by two times the input bit duration to form 2-bit symbols. There is also an inherent half a symbol time delay in Q channel with respect to the I channel at the output of this block. This block can thus be used in Offset QPSK and MSK implementations.**

Inputs & Outputs **Input *x* represents CCSK encoded data bits. Outputs *real* and *imag* represent I and Q rails of symbols.**

Parameters **None**

Name **MSK MODULATOR** */caedata/chaoslib/msk_mod*



MSK MODULATOR BLOCK PARAMETERS	
MAIN PARAMETERS:	
Samples per Symbol (integer)	8
Initial Delay (integer)	4

Type Primitive

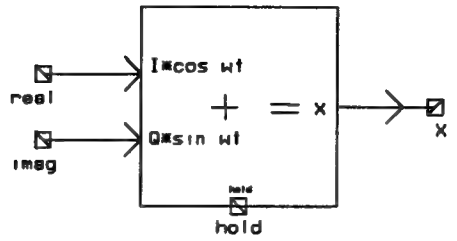
Description This block implements the basic half-sinusoidal pulse shaping functions involved in MSK modulation. It multiplies the in-phase and quadrature-phase components of the baseband symbol by half sine waves thus providing pulse shaping. In order to improve computational efficiency, the values of the sine and the cosine functions are computed during intialization stored. In the run phase, these stored values are looked up. Since the samples are evenly spaced discrete samples of a periodic function, the values repeat themselves and thus look up tables cut down on computations dramatically during the run phase of the code.

Inputs & Outputs Inputs x and y represent I and Q rails of input data respectively. Outputs *real* and *imag* represent half-sinusoid weighted input symbols.

Parameters

- **Samples per symbol:** number of samples per symbol which is twice the number of samples per information bit at the input of the data_iq block.
- **Initial delay:** This is the offset in samples in the Q channel with respect to the I channel

Name **I*COS(wt) + Q*SIN(wt) = X** */caedata/chaoslib/carrier*



IF CARRIER BLOCK PARAMETERS	
MAIN PARAMETERS:	
Samples per Period (integer)	8
Phase Offset (of In phase)	0.0

Type Primitive

Description This block implements the IF carrier modulation portion of MSK modulation. It multiplies the in-phase component with a cosine carrier and the quadrature component with a sine carrier and adds the two waveforms to form the passband MSK waveform.

In order to improve computational efficiency, the values of the sine and the cosine functions are computed during intialization stored. In the run phase, these stored values are looked up. Since the samples are evenly spaced discrete samples of a periodic function, the values repeat themselves and thus look up tables cut down on computations dramatically during the run phase of the code.

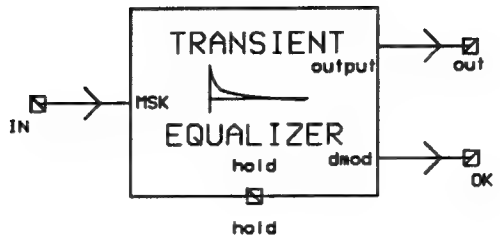
Inputs & Outputs Inputs ***real*** and ***imag*** represent pulse shaped I and Q rails of input data. Output x represents the MSK waveform at IF.

Parameters

- **Samples per period:** Number of samples per period of the carrier which is the same as the number of samples per symbol for the case of IF rate = symbol rate.
- **Phase offset (in radians):** Initial phase offset in carrier.

Receiver

Name **TRANSIENT EQUALIZER** */caedata/chaoslib/adequalize*



TRANSIENT DETECTOR PARAMETERS	
Feed through type	ALL_FEED_THROUGH
Signal Power Threshold	0.16
Samples per Symbol Ls	16
Av. over no. symbols	3

Type **Primitive**

Description **The annoying transient response of some A/D boards (particularly the Spectrum DB212) can be detected by this block. The output signal is 0 (low) when the transients are over and is a 1 (high) until that time. This block estimates the DC value in the transient signal as a moving average of a window of symbols specified as a parameter. It then subtracts this estimated DC value from the middle sample of the window, which is the current sample under consideration. The final decision as to whether the transients are over is made by testing whether the signal power crosses a threshold specified as parameter.**

The block uses two contiguous buffers to store the input samples. The second buffer is a mirror image of the first one and it helps to reduce the number of pointer comparisons. The current element pointer of the buffer is offset by half the buffer length and thus in the absence of the second buffer, one has to check the value of the pointer before incrementing it so that buffer overflow problems are avoided. By having an image of the buffer

in the immediate address space that follows the buffer, the current pointer buffer can be allowed to go into the second buffer with no overflow problems.

Whenever the power level falls below the threshold, the block raises the hold signal (which follows reverse logic) and indicates to the carrier recovery and the demodulator blocks to hold all computations. These blocks in turn hold all the blocks that follow them in the receiver. There is a delay however between the time that the power level drops and the time that this block detects that drop. The delay is caused by the buffering of the input samples.

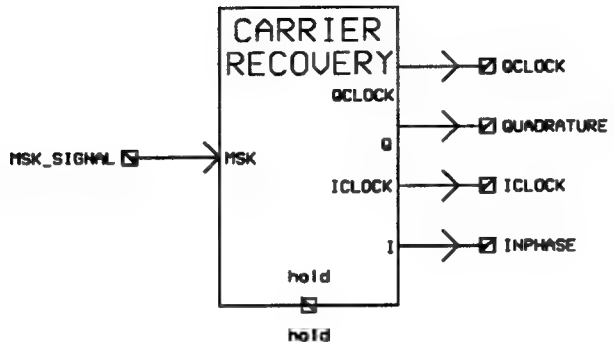
Inputs & Outputs

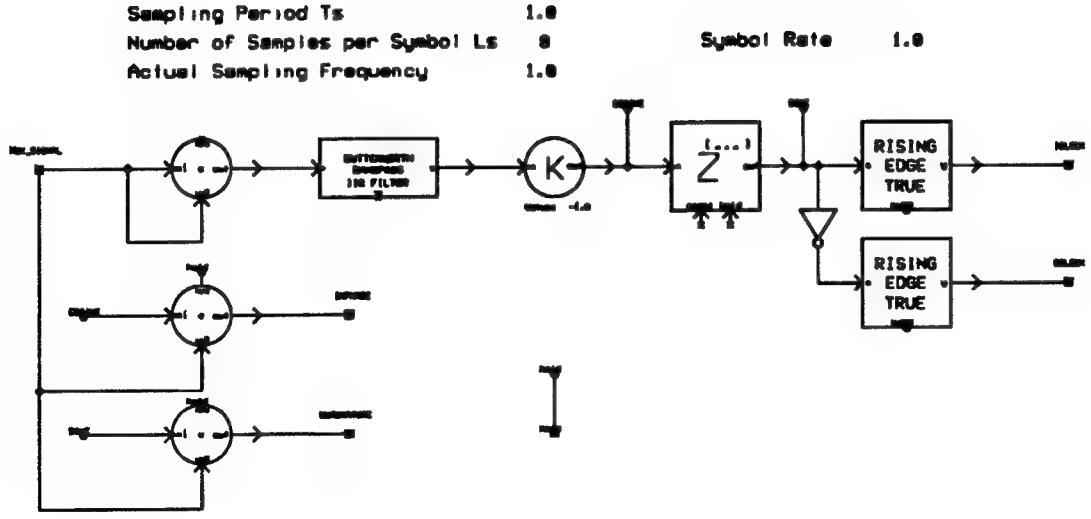
Input *IN* is the received MSK signal. Output *OK* indicates the presence or absence of transients while output *out* represents the equalized output to be sent for demodulation.

Parameters

- **Average number of symbols:** Number of symbols over which moving average is performed
- **Signal power threshold:** threshold to be crossed by signal power for the block to detect presence of signal
- **Number of samples per symbol:** Number of samples per symbol duration

Name **CARRIER RECOVERY** */caedata/chaoslib/carrier_rec*





Type Detailed

Description This detail implements the carrier recovery portion of the MSK demodulation process. It implements a modified version of the MSK self-synchronization algorithm. For the specific case of sampled data and IF rate = symbol rate, this algorithm reduces to squaring the incoming signal and passing it through a bandpass filter having center frequency the same as carrier frequency and and an arbitrarily narrow bandwidth. When specifying the center frequency for the parameter of the digital bandpass filter, the value specified should be the carrier frequency normalized by the sampling frequency of the system. It is important that the sampling frequency used for normalization should be the actual sampling frequency set by the hardware (when doing real-time implementation). This algorithm also extracts the clock from the carrier by hard-limiting.

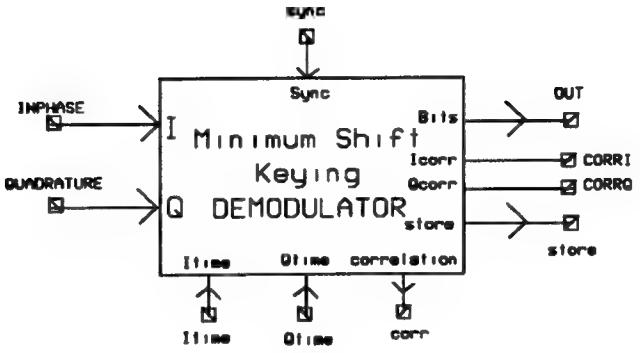
Inputs & Outputs Input *MSK* represents the equalized input signal from the transient detector. Outputs *ICLOCK* and *QCLOCK* represent symbol timing for the I and Q channels respectively. Outputs *INPHASE* and *QUADRATURE* represent the baseband inphase and quadrature components after the carrier is removed. The input *hold* may be used to hold the block.

Parameters

- **Samples per symbol:** Number of samples per MSK symbol
- **Center frequency:** Normalized center frequency of bandpass filter which is the same as carrier frequency
- **Passband Bandwidth:** Normalized bandwidth of bandpass filter
- **Actual sampling frequency:** Sampling frequency set by hardware which may not be exactly equal to the user-specified sampling frequency

Other parameters are not currently being used.

Name **DEMODULATOR MSK** */caedata/chaoslib/msk_demod*



Block Parameters	
Number of Samples per Symbol, Ls	8
FEED_TYPE	ALL_FEED_THROUGH

Type **Primitive**

Description **This block performs the the baseband correlate and dump function in the I and Q channels for MSK demodulation. When there is no energy present at the input to the receiver, this block outputs a store signal value of -4.0. The block is first turned on when it receives the first I clock. At this point it starts correlating the samples in the I channel by performing one multiply accumulate operation per sample . At the same time it keeps outputting a store signal is -3. Approximately half the time before it receives the next I clock, it receives the Q clock and starts correlation operation for the Q channel also. When it receives the next I clock, it makes a decision whether the accumulated I correlation value is positive or negative and hence a 1 or 0. At this point it outputs the bit value , makes the store signal 1, clears the I correlation buffer to 0 and starts a new I correlation. For the next two**

samples, it outputs store signal value of -1 and the fourth sample it makes the store signal -3. Next sample if it receives the next Q clock, it makes a decision whether the accumulated Q correlation value is positive or negative (and hence a 1 or a 0) , outputs the bit value, makes the store signal 1, clears the Q correlation buffer to 0 and starts a new Q correlation. Else it continues outputting a store signal value of -3. Thus, a store signal value of 1 indicates that new bit decision, whether I or Q, has been made.

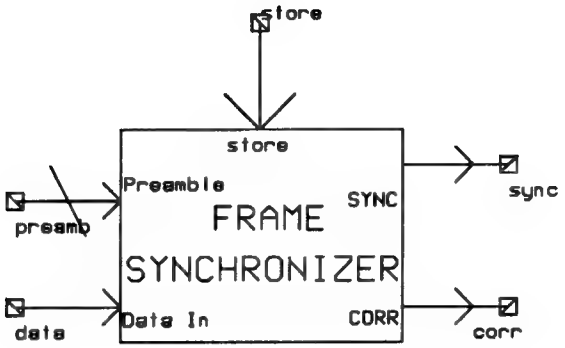
Inputs & Outputs

Input *INPHASE* represents the inphase component of received MSK signal after the carrier is removed while input *QUADRATURE* represents the quadrature component. Inputs *Itime* and *Qtime* represent the I and Q clocks extracted by the clock recovery algorithm. The *sync* input when connected to the *OK* output of the transient detector disables or enables the block in accordance to OK equals 0 or 1. Outputs *CORRI* and *CORRQ* indicate the correlation values for the I and Q channels respectively. Output *OUT* represents the demodulated bits whereas output *store* indicates presence of a new bit.

Parameters

- **Number of samples per symbol:** Number of samples per symbol duration

Name **FRAME SYNCHRONIZER** */caedata/chaoslib/frame_sync2*



FRAME SYNC PARAMETERS	
Feed through type	ALL_FEED_THROUGH
Preamble Size in Bits	32
Preamble Sequence	2095647468
Vector Length	32
Threshold	26
Samples per Symbol	8
Number of sync bursts	6
Packet Size in Encoded Bits	1200

Type **Primitive**

Description **This block performs frame synchronization. The preamble sequence used, size of the preamble sequence in bits, number of preamble sequences at the beginning of each packet, packet size in CCSK encoded bits and number of samples per symbol may all be specified as parameters. Currently, the block is optimized only for 8 and 16 samples per symbol. There is another parameter called threshold which represents the minimum value of the correlation , between received preamble sequence and the stored preamble sequence, required for achieving frame synchronization. There is also provision for specifying the preamble sequence as a vector connected to the preamb port of the block. The vector length may be specified as a parameter.**

The criterion for synchronization used in this block is that the correlation of the input data bits and the synchronization sequence exceed the specified threshold every single time the sequence is sent. In other words if the sequence is sent N times then the threshold has to be crossed N times as well. During the synchronization, the block divides its operations over all the samples in the bit. Thus it performs partial correlations each sample.

This block is also responsible for keeping track of the packets in the receiver, i.e. it detects the beginning of a packet and synchronizes to the packet's frame sequence.

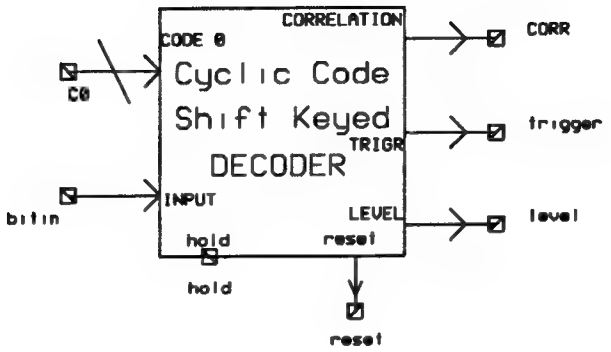
Inputs & Outputs

Input *data* represents demodulated bit stream while input *preamb* indicates an optional vector port for user-specified preamble sequence. Input *store* represents the presence of a new bit. Output *sync* is nothing but the input store signal inverted in sign. Output *corr* represents the correlation value for the current window of length equal to preamble sequence length.

Parameters

- **Number of samples per symbol:** Number of samples per symbol duration
- **Preamble size in bits:** size of preamble sequence in bits
- **Preamble sequence:** decimal equivalent of desired preamble sequence
- **Packet size in encoded bits:** size of each packet in terms of CCSK encoded levels
- **Number of frame sync sequences:** number of preamble seque nces transmitted at the beginning of each packet

Name **DECODER CCSK** */caedata/chaoslib/decoder2*



DECODER BLOCK PARAMETERS	
Feed through type	ALL_FEED_THROUGH
Code Size in Bits	32
Block Code of 0	2095647468
Code Vector Size	32
Samples per Symbol	8
Levels per packet	300

Type **Primitive**

Description **This block performs 32 to 5 CCSK decoding. The codeword size in bits, zero level codeword, number of samples per symbol and number of CCSK levels per packet of input data may all be specified as parameters. There is also a provision for specifying the zero level codeword through a vector input port. This block decides as to which codeword was transmitted by correlating the received codeword with all the codewords in the CCSK look-up table and decides in favor of that codeword which yields maximum correlation. When frame synchronization has not been achieved this block outputs a level 31 which is decoded as stop bits by the CCSK to RS232 block.The sync signal from the frame synchronization block may be used to hold this block. This block performs computations only when the hold signal takes on values less**

than or equal to 2.0 i.e. when the sync signal takes on values -1 and 1. Currently, the block is optimized only for 8 and 16 samples per symbol. There is a latency of one codeword in this block i.e. when samples of the last bit in the input codeword are being received, this block finishes with computing the correlation of the input codeword with all the entries in the CCSK look-up table, but it still has to find out the maximum of these correlation values. This it does when the next CCSK codeword is being received.

Also there is one sample delay in the sync signal at the output of the frame sync block with respect to the data bit samples at the input to the frame sync block. Thus, if the data bits input to the frame sync block is also used as the data bits input to the CCSK decoder, then a unit delay block should be used at the data bits input to the CCSK decoder.

In order to expedite the computations and to distribute them evenly among the samples in the bit, this block performs partial correlations each sample. It also uses two buffers and switches the buffer pointers every time a new code word begins. This distributes the number of cycles spent to reinitialize the correlations evenly among all the bits in the code word.

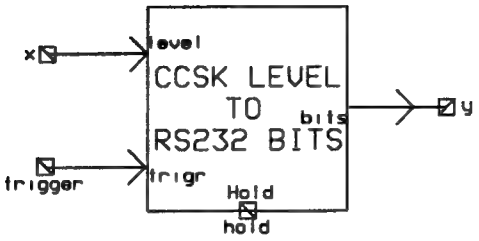
Inputs & Outputs

Input *bitin* represents demodulated bit stream, while input *C0* represents optional vector port for user-specified zero level CCSK codeword. Input *hold* may be used to hold the computations in the block. Output *corr* represents the maximum correlation value for the previous received codeword. Output *level* represents the level corresponding to 5 bits obtained by decoding a 32-bit codeword, while output *trigger* is 1 when there is a level to be output and 0 otherwise. Output *reset* is not currently being used.

Parameters

- **Number of samples per symbol:** Number of samples per symbol duration
- **Codeword size in bits:** size of codeword in bits
- **Block code of zero** decimal equivalent of zero level codeword
- **Levels per packet:** size of each packet in terms of CCSK encoded levels

Name **CCSK LEVEL TO RS232 BITS** */caedata/chaoslib/level2bits*



LEVEL2BITS BLOCK PARAMETERS	
Feed through type	ALL_FEED_THROUGH
Precision (number of bits)	5
Data Rate	300.0
Input Samping Rate	9600.0
Logic (1 for positive logic, 0 for negative)	1

Type **Primitive**

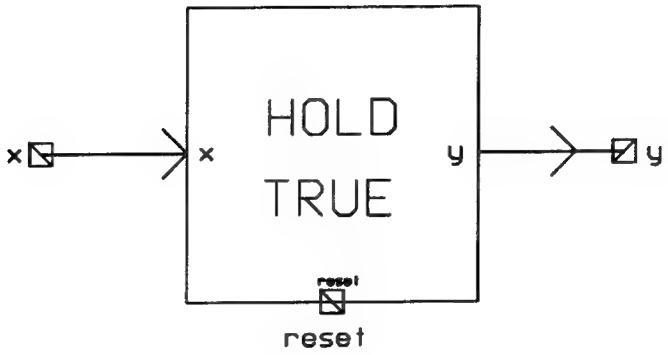
Description This block takes CCSK levels (decimal) and outputs the binary bit equivalent of these levels. The information bit rate and sampling frequency may be set as parameters. Other parameters are not currently being used.

Inputs & Outputs Input *x* represents CCSK levels from the decoder while input *trigger* represents signal from decoder which indicates presence of a new level. Input *hold* may be used to hold the block. Output *y* represents the binary bit equivalent of the input level

Parameters

- **Number of samples per symbol:** Number of samples per symbol duration
- **Bit rate:** information bit rate

Name **HOLD TRUE** */caedata/chaoslib/hold_true*



HOLD TRUE VALUE PARAMETERS	
Feed through type	ALL_FEED_THROUGH

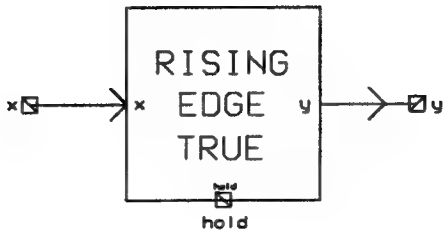
Type Primitive

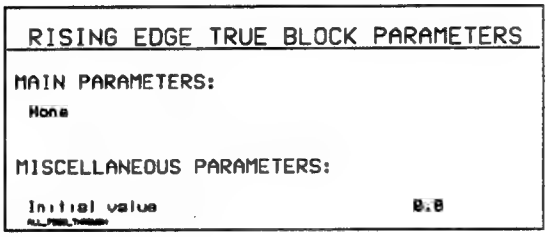
Description To begin with, this block outputs a -1. When the x input to the block is ¿ 0.0, the block outputs a 1 at its y output. When the reset input is ¿ 0.0, this block then outputs -1 at its y output

Inputs & Outputs This block has one input x apart from the *hold* input and one output y.

Parameters None

Name **RISING EDGE TRUE** */caedata/chaoslib/rise_edge_true*





Type **Primitive**

Description **This block functions as a hard limiter. Whenever the input makes a transition from zero or negative to a positive value, this block outputs a 1, otherwise it outputs a 0. The hold input may be used to reset the output value to the initial value parameter. This happens whenever the hold input is greater than 0.**

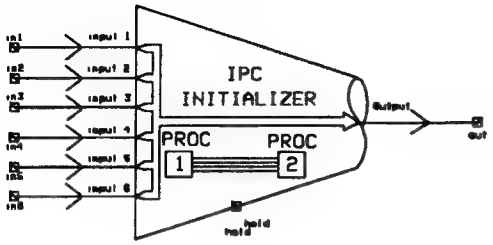
Inputs & Outputs **This block has two inputs, x which is the input that needs to be hard-limited and *hold* which is used as a reset input. The hard-limited output is y.**

Parameters

- **Initial value:** Initial value of output

Interprocessor Communication Blocks

Name **IPC1-IN INITIALIZER** */caedata/chaoslib/ipc1_in*



IPC MULTIPLEXER FOR C40	
Feed through type	ALL_FEED_THROUGH

Type Primitive

Description The suggested method IPC-1 is partly implemented in this block. The block can "multiplex" a maximum of six input signals. During the initialization of this block, it sets up the autoinitialization sequence of the particular DMA channel that can be synchronized to the communication port. The DMA channel index is obtained from the *cp_con* matrix in the "mytypes.h" file. The linked list of autoinitialization sequences for the DMA channel is either created by the block (if it is the first block that transfers data to the receiving C40) or it attaches another pair of autoinitialization sequences to the linked list. In either case the block creates two autoinitialization sequences. Since this is the transmitting C40, the first sequence sets up the DMA channel to write the data to the comm. port. The sequence is set up so that during the actual run phase, the DMA autoinitializes to the second sequence as soon as it finishes the data transfer. The second sequence sets up the DMA to receive the acknowledgement from the receiving processor. The acknowledgement received is actually the receipt for the previous transfer. After a successful receipt of the acknowledgement, the DMA channel halts and waits for the CPU to restart it. During the entire operation of the DMA channel, the CPU is given access priority over the DMA. This is done since the actual transfer itself takes very little time compared to the CPU execution time (per iteration). Experience has shown that giving priority to the CPU speeds up the algorithm by a significant factor.

The input data is buffered into a common buffer called *DMABuf* that is used to make the transfer (since the DMA channel cannot transfer arbitrarily placed address contents). The transfer counter is set to one more than the number of data words and so is the buffer length. The last transfer is used by the receiving processor to determine whether the data in its corresponding buffer is new or stale.

Additionally this block also sets up the DIE register (a CPU primary register) so that the specific DMA channel is synchronized to interrupts generated by the appropriate communication port.

Inputs & Outputs

There are a maximum of six input ports that have to be connected starting from the top, i.e. if there are three signals shared by the two processors, then make sure that input ports 1, 2 and 3 are connected (counting from top) and 4, 5, 6 are disconnected. The output port of the block is a dummy signal and should only be connected to an *ipc1_out* block. All the input ports are scalar.

Parameters

- **Feed Through Type:** ALL_FEED_THROUGH. All inputs need to valid before the output can be calculated (uneditable).
- **P_fromproc:** The processor ID (0, 1, etc.) of this processor.
- **P_toproc:** The processor ID (0, 1, etc.) of the receiving processor (that is connected to this C40).

Error conditions

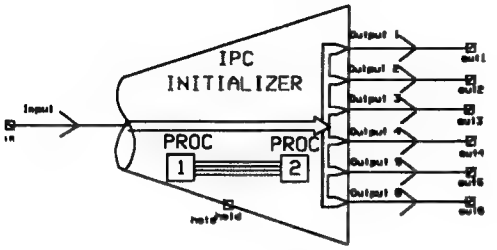
The following lists the expected type of each parameter. Parameter Entry Errors occur if there is a type mismatch.

1. **P_fromproc, P_toproc:** Must be a positive integer (0 onwards).

See Also

The output block IPC1-OUT INITIALIZER, and also the IPC-2 blocks.

Name **IPC1-OUT INITIALIZER** */caedata/chaoslib/ipc1_out*



IPC MULTIPLEXER FOR C40	
Feed through type	ALL_FEED_THROUGH

Type Primitive

Description The suggested method IPC-1 is partly implemented in this block. The block has a dummy input and can demultiplex the actual received data into a maximum of six output signals. During the initialization of this block, it sets up the autoinitialization sequence of the particular DMA channel that can be synchronized to the communication port. The DMA channel index is obtained from the *cp_con* matrix in the "mytypes.h" file. The linked list of autoinitialization sequences for the DMA channel is either created by the block (if it is the first block that receives data from the appropriate C40) or it attaches another pair of autoinitialization sequences to the linked list. In either case the block creates two autoinitialization sequences. Since this is the receiving C40 (as far as this block operation goes), the first sequence sets up the DMA channel to write the acknowledgement for the previous transfer to the comm. port. The sequence is set up so that during the actual run phase, the DMA autoinitializes to the second sequence as soon as it finishes the acknowledgement transfer. The second sequence sets up the DMA to receive the actual data from the transmitting processor. After a successful data transfer operation, the DMA channel halts and waits for the CPU to restart it. During the entire operation of the DMA channel, the CPU is given access priority over the DMA. This is done since the actual transfer itself takes very little time compared to the CPU execution time (per iteration). Experience has shown that giving priority to the CPU speeds up the algorithm by a significant factor.

The input data is buffered into a common buffer called *DMABuf* that is used to make the transfer (since the DMA channel cannot transfer to arbitrarily placed address locations). The transfer counter is set to one more than the number of data words and so is the buffer length. The last transfer is used by the processor to determine whether the data in its corresponding buffer is new or stale.

Additionally this block also sets up the DIE register (a CPU primary register) so that the specific DMA channel is synchronized to interrupts generated

by the appropriate communication port.

Inputs & Outputs

There are a maximum of six output ports that have to be connected starting from the top, i.e. if there are three signals shared by the two processors, then make sure that output ports 1, 2 and 3 are connected (counting from top) and 4, 5, 6 are disconnected. The input port of the block is a dummy signal and should only be connected to an *ipc1_in* block. All the output ports are scalar.

Parameters

- **Feed Through Type:** ALL_FEED_THROUGH. All inputs need to valid before the output can be calculated (uneditable).
- **P_fromproc:** The processor ID (0, 1, etc.) of the transmitting processor (that is connected to this C40).
- **P_toproc:** The processor ID (0, 1, etc.) of this processor.

Error conditions

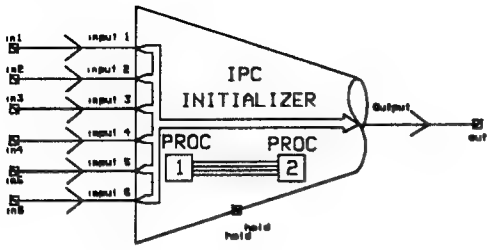
The following lists the expected type of each parameter. Parameter Entry Errors occur if there is a type mismatch.

1. **P_fromproc, P_toproc:** Must be a positive integer (0 onwards).

See Also

The output block IPC1-IN INITIALIZER, and also the IPC-2 blocks.

Name **IPC2-IN INITIALIZER** */caedata/chaoslib/ipc2_in*



IPC MULTIPLEXER FOR C40	
Feed through type	ALL_FEED_THROUGH

Type Primitive

Description The suggested method IPC-2 is partly implemented in this block. The block can "multiplex" a maximum of six input signals. During the initialization of this block, it sets up the autoinitialization sequence of the particular DMA channel that can be synchronized to the communication port. The DMA channel index is obtained from the *cp_con* matrix in the "mytypes.h" file. The linked list of autoinitialization sequences for the DMA channel is either created by the block (if it is the first block that transfers data to the receiving C40) or it attaches another autoinitialization sequences to the linked list. In either case the block creates only one autoinitialization sequence. Since this is the transmitting C40, the sequence sets up the DMA channel to write the data to the comm. port. After a successful data transfer operation, the DMA raises an interrupt flag in its control register, sets the START bits to 01 and sets the transfer counter to 0. and then it halts and waits for the CPU to restart it. During the entire operation of the DMA channel, the CPU is given access priority over the DMA. This is done since the actual transfer itself takes very little time compared to the CPU execution time (per iteration). Experience has shown that giving priority to the CPU speeds up the algorithm by a significant factor. When the CPU is ready with the next data, it polls the control register at the appropriate locations to determine if the previous transfer was successfully completed. Due to the reduced number of transfers as compared with IPC-1, this algorithm performs slightly faster than IPC-1.

The input data is buffered into a common buffer called *DMABuf* that is used to make the transfer (since the DMA channel cannot transfer arbitrarily placed address contents). The transfer counter is set to the number of data words and so is the buffer length.

Additionally this block also sets up the DIE register (a CPU primary register) so that the specific DMA channel is synchronized to interrupts generated by the appropriate communication port.

Inputs & Outputs

There are a maximum of six input ports that have to be connected starting from the top, i.e. if there are three signals shared by the two processors, then make sure that input ports 1, 2 and 3 are connected (counting from top) and 4, 5, 6 are disconnected. The output port of the block is a dummy signal and should only be connected to an *ipc1_out* block. All the input ports are scalar.

Parameters

- **Feed Through Type:** ALL_FEED_THROUGH. All inputs need to valid before the output can be calculated (uneditable).
- **P_fromproc:** The processor ID (0, 1, etc.) of this processor.
- **P_toproc:** The processor ID (0, 1, etc.) of the receiving processor (that is connected to this C40).

Error conditions

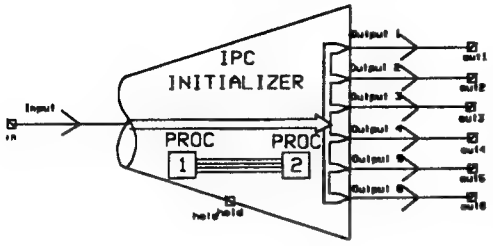
The following lists the expected type of each parameter. Parameter Entry Errors occur if there is a type mismatch.

1. **P_fromproc, P_toproc:** Must be a positive integer (0 onwards).

See Also

The output block IPC2-OUT INITIALIZER, and also the IPC-1 blocks.

Name **IPC2-OUT INITIALIZER** */caedata/chaoslib/ipc2_out*



IPC MULTIPLEXER FOR C40	
Feed through type	ALL_FEED_THROUGH

Type Primitive

Description The suggested method IPC-2 is partly implemented in this block. The block has a dummy input and can demultiplex the actual received data into a maximum of six output signals. During the initialization of this block, it sets up the autoinitialization sequence of the particular DMA channel that can be synchronized to the communication port. The DMA channel index is obtained from the *cp_con* matrix in the "mytypes.h" file. The linked list of autoinitialization sequences for the DMA channel is either created by the block (if it is the first block that receives data from the transmitting C40) or it attaches another autoinitialization sequences to the linked list. In either case the block creates only one autoinitialization sequence. Since this is the receiving C40, the sequence sets up the DMA channel to read the data from the comm. port. After a successful data transfer operation, the DMA raises an interrupt flag in its control register, sets the START bits to 01 and sets the transfer counter to 0. and then it halts and waits for the CPU to restart it. During the entire operation of the DMA channel, the CPU is given access priority over the DMA. This is done since the actual transfer itself takes very little time compared to the CPU execution time (per iteration). Experience has shown that giving priority to the CPU speeds up the algorithm by a significant factor. When the CPU is ready to read the next data, it polls the control register at the appropriate locations to determine if the previous transfer was successfully completed. Due to the reduced number of transfers as compared with IPC-1, this algorithm performs slightly faster than IPC-1.

The output data is buffered into a common buffer called *DMABuf* that is used to make the transfer (since the DMA channel cannot write to arbitrarily placed address locations). The transfer counter is set to the number of data words and so is the buffer length.

Additionally this block also sets up the DIE register (a CPU primary register) so that the specific DMA channel is synchronized to interrupts generated by the appropriate communication port.

Inputs & Outputs

There are a maximum of six output ports that have to be connected starting from the top, i.e. if there are three signals shared by the two processors, then make sure that output ports 1, 2 and 3 are connected (counting from top) and 4, 5, 6 are disconnected. The input port of the block is a dummy signal and should only be connected to an *ipc1_in* block. All the output ports are scalar.

Parameters

- **Feed Through Type:** ALL_FEED_THROUGH. All inputs need to valid before the output can be calculated (uneditable).
- **P_fromproc:** The processor ID (0, 1, etc.) of the transmitting processor (that is connected to this C40).
- **P_toproc:** The processor ID (0, 1, etc.) of this processor.

Error conditions

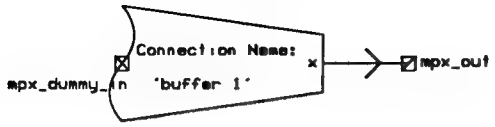
The following lists the expected type of each parameter. Parameter Entry Errors occur if there is a type mismatch.

1. **P_fromproc, P_toproc:** Must be a positive integer (0 onwards).

See Also

The output block IPC2-IN INITIALIZER, and also the IPC-1 blocks.

Name **NULLSIB** */caedata/chaoslib/nullsib*



SCALAR INPUT BLOCK PARAMETERS	
Feed through type	ALL_FEED_THROUGH

Type Primitive

Description This is a dummy block that Multiprox inserts in the path of each signal that goes between two processors. The companion block that this connects to, is the ***nullsob*** block. The block connects to the dummy output signal of the ***ipc1_in*** or the ***ipc2_in*** block.

Inputs & Outputs Both the input and the output ports of this block carry dummy signals and there is no operation performed in the block.

Parameters

- **Feed Through Type:** ALL_FEED_THROUGH. All inputs need to valid before the output can be calculated (uneditable).
- **Buffer Length:** Length of the buffer inside (uneditable). Kept to keep Multiprox from getting confused.

Error conditions none

See Also ***nullsob*** block that this one connects with.

Name **NULLSOB** */caedata/chaoslib/nullsob*

SCALAR OUTPUT BLOCK PARAMETERS	
Feed through type	ALL_FEED_THROUGH

Type Primitive

Description This is a dummy block that Multiprox inserts in the path of each signal that goes between two processors. The companion block that this connects to, is the ***nullsib*** block. The block connects to the dummy input signal of the *ipc1_out* or the *ipc2_out* block.

Inputs & Outputs Both the input and the output ports of this block carry dummy signals and there is no operation performed in the block.

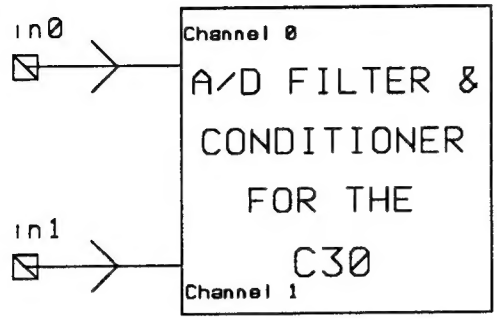
Parameters

- **Feed Through Type:** ALL_FEED_THROUGH. All inputs need to valid before the output can be calculated (uneditable).
- **Buffer Length:** Length of the buffer inside (uneditable). Kept to keep Multiprox from getting confused.

Error conditions none

See Also ***nullsib*** block that this one connects with.

Name **ATOD FILTER AND TIMER** */caedata/chaoslib/atod_filt*



A to D DATA CONDITIONER PARAMETERS	
Desired Sampling Frequency	9600.0
Condition the Data ? 'yes' or 'no'. If 'yes'	'yes'
Interpolating Algorithm 'linear' or 'quadratic'	'linear'

Type Primitive

Description The purpose of this block is to time every sampling interval (or iteration) in SPW. This is done using the timers on the C30 or the C40 processor. When the block determines that there is time to spare (i.e. the processing prior to this block finished earlier), it can interpolate the samples and send them to the digital to analog converter. It can also perform any other operation that the user might need to be done till the sampling interval epoch is over. At present however code only exists to linearly interpolate the input samples and output a variable number of samples to the DAC.

The major problem with this block is that the sampling interval can only be accurate to within the smallest sampling interval possible with the DAC. While the block does not accumulate timing errors, the clock drifts mentioned above cause problems in sensitive sections of the receiving processor algorithm (that is connected to the other end of the DAC link). Consider for instance the carrier and clock recovery algorithm in the digital radio. The clock drifts are sufficiently large that the timing obtained for demodulation is all inaccurate.

So this block should only be used for smoothing the DAC output in more or less rugged (or insensitive) applications.

Inputs & Outputs The input carries the two signals that need to be interpolated and possibly sent out to the digital to analog converter. There is no output port of the block.

Parameters

- **Feed Through Type:** ALL_FEED_THROUGH. All inputs need to valid before the output can be calculated (uneditable).
- **P_proc:** 'C30' or 'C40'.
- **P_query:** "yes" if the user wants to interpolate the data and send it to

the D/A converter, "no" if the user is only interested in the time left for processing in that sampling duration (not used at present).

- **P_sfreq:** The actual sampling frequency desired.
- **P_type:** Not used (leave as 'linear').

Error conditions

The following lists the expected type of each parameter. Parameter Entry Errors occur if there is a type mismatch.

1. **P_sfreq:** Must be a positive real number.
2. **P_proc:** Has to be either 'C30' or 'C40'.

☆U.S. GOVERNMENT PRINTING OFFICE: 1995-610-126-50181

MISSION

OF

ROME LABORATORY

Mission. The mission of Rome Laboratory is to advance the science and technologies of command, control, communications and intelligence and to transition them into systems to meet customer needs. To achieve this, Rome Lab:

a. Conducts vigorous research, development and test programs in all applicable technologies;

b. Transitions technology to current and future systems to improve operational capability, readiness, and supportability;

c. Provides a full range of technical support to Air Force Materiel Command product centers and other Air Force organizations;

d. Promotes transfer of technology to the private sector;

e. Maintains leading edge technological expertise in the areas of surveillance, communications, command and control, intelligence, reliability science, electro-magnetic technology, photonics, signal processing, and computational science.

The thrust areas of technical competence include: Surveillance, Communications, Command and Control, Intelligence, Signal Processing, Computer Science and Technology, Electromagnetic Technology, Photonics and Reliability Sciences.